DAVI EINSTEIN MELGES ARNAUT

PHOENIX - UM COMPONENTE RELACIONAL PARA PLATAFORMAS DE ARMAZENAMENTO EM NUVEM

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.

Orientadora: Profa. Dra. Carmem Satie Hara

CURITIBA

2010

DAVI EINSTEIN MELGES ARNAUT

PHOENIX - UM COMPONENTE RELACIONAL PARA PLATAFORMAS DE ARMAZENAMENTO EM NUVEM

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.

Orientadora: Profa. Dra. Carmem Satie Hara

CURITIBA

2010

Ministério da Educação
Universidade Federal do Paraná
Programa de Pós-Graduação em Informática

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática, do aluno Davi Einstein Melges Arnaut, avaliamos o trabalho intitulado, *"UM COMPONENTE RELACIONAL PARA PLATAFORMAS DE ARMAZENAMENTO EM NUVEM"*, cuja defesa foi realizada no dia 27 de agosto de 2010, às 21:00 horas, Auditório do Departamento de Informática do Setor de Ciências Exatas da Universidade Federal do Paraná. Após a avaliação, decidimos pela aprovação do candidato.

Curitiba, 27 de agosto de 2010.

Profª. Dra. Carmem Satie Hara
DINF/UFPR – Orientador

Prof. Dr. Vidal Martins
PUCPR – Membro Externo

Prof. Dr. Eduardo Cunha de Almeida
DINF/UFPR – Membro Interno

AGRADECIMENTOS

Agradeço especialmente minha orientadora, professora Carmem Satie Hara, por sua paciência e afinco durante os mais de dois anos de trabalho neste projeto.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

O crescente volume e diversidade de dados, juntamente com a evolução dos conceitos da computação em nuvem, têm dado origem a um novo tipo de sistemas de banco de dados distribuído com foco em escalabilidade à custa de características comumente associadas a bancos de dados relacionais tradicionais. Na maioria dos casos, estes novos sistemas são destinados a implantação em larga escala e escalabilidade maciça, nos quais as características tradicionais de bancos de dados relacionais acabam por reduzir sua viabilidade como uma plataforma distribuída de armazenamento de dados.

Para abordar esta questão, este trabalho propõe um novo design e arquitetura para um sistema de banco de dados relacional baseado em nuvem. O componente central deste sistema é um módulo de armazenamento, que é responsável por mapear o esquema lógico, baseado em relações, para um esquema físico, baseado em um repositório de dados distribuído. A arquitetura estratificada proposta oferece independência de dados física, permitindo que diferentes abordagens para mapeamento de dados e fragmentação, enquanto o repositório de dados distribuído é responsável por fornecer escalabilidade, disponibilidade, replicação de dados e propriedades ACID.

Um protótipo do sistema, chamado de Phoenix, foi desenvolvido com base na arquitetura proposta usando um repositório de dados transacional. Um estudo experimental realizado em um grupo de servidores genéricos mostra que a Phoenix preserva as propriedades desejadas do repositório de dados, ao fornecer a funcionalidade de banco de dados relacional sem decorrer em uma alta sobrecarga.

ABSTRACT

The ever-increasing volume and diversity of data coupled with the dissemination and maturation of various concepts of cloud computing has given rise to a new type of distributed database systems with a focus on scalability at the cost of benefits associated with traditional relational databases. These new systems are often aimed at large-scale deployment and massive scalability. In such scenarios, features of relational databases end up reducing its viability as a platform for distributed data storage.

To address this issue, this work proposes a new design and architecture of a cloud-based relational database system. The system's core component is a storage engine, which is responsible for mapping the logical schema, based on relations, to a physical storage, based on a distributed key-value datastore. The proposed stratified architecture provides physical data independence, by allowing different approaches for data mapping and partitioning, while the distributed datastore is responsible for providing scalability, availability, data replication and ACID properties.

A prototype of the system, named Phoenix, has been developed based on the proposed architecture using a transactional key-value store. An experimental study on a cluster of commodity servers shows that Phoenix preserves the desired properties of key-value stores, while providing relational database functionality at a very low overhead.

CAPÍTULO 1

INTRODUÇÃO

O amadurecimento dos diversos conceitos de computação em nuvem vem mudando os aspectos econômicos da computação, principalmente devido à introdução de serviços baseados em dados com um custo acessível, sob demanda e em tempo real [Armbrust et al., 2009]. Um semblante desta mudança de paradigma se deve a novas formas de manusear e disponibilizar dados através de arquiteturas distribuídas e baseadas em serviços, de modo que os dados possam ser acessados de forma fácil e ubíqua [Buyya et al., 2008]. Esse novo modelo de computação habilita uma escalabilidade maciça de serviços e realça oportunidades de colaboração, integração e análise sobre uma plataforma comum e compartilhada.

Para viabilizar e dar suporte a esse novo modelo surgiu a necessidade de um novo tipo de sistema de banco de dados centrado em escalabilidade à custa de benefícios dos tradicionais bancos de dados relacionais. Apesar de um Sistema Gerenciador de Banco de Dados Relacional (SGBDR) tradicional oferecer uma mistura de simplicidade, flexibilidade, robustez e confiabilidade, algumas dessas funcionalidades implicam aumento significativo da complexidade quando se tenta assegurá-las em um sistema distribuído em centenas ou milhares de elementos computacionais (nós ou nodos). Em arquiteturas deste tipo, as características que tornam um SGBDR tão atraente acabam reduzindo drasticamente sua viabilidade como plataforma de armazenamento de dados.

Este novo tipo de sistema gerenciador é comumente chamado de banco de dados chave-valor (*key/value datastore*), mas também é definido como serviço de armazenamento de dados distribuído ou em nuvem. Existem diversos sistemas desenvolvidos recentemente com o propósito de explorar serviços de armazenamento na nuvem, como o Amazon SimpleDB [Amazon, 2007, DeCandia et al., 2007], Google App Engine [Google, 2008, Chang et al., 2006], Windows Azure [Microsoft, 2010], Cassandra [Lakshman and Malik, 2010], PNUTS [Cooper et al., 2008], e G-Store [Das et al., 2010]. Todos fornecem um serviço semelhante: uma interface simples para o armazenamento de tuplas que permite a inserção, resgate e remoção de seus elementos individualmente. Além da similaridade

entre as interfaces de serviço, as propriedades de escalabilidade e disponibilidade são comumente alcançadas através de uma síntese de técnicas *peer-to-peer* (P2P), tais como auto-organização, descentralização e balanceamento de carga.

A maneira disseminada de composição dessas técnicas é a utilização de uma rede P2P estruturada baseada em Tabela de Dispersão Distribuída (*Distributed Hash Table* - DHT) para prover uma interface de uso geral para localização, armazenamento e endereçamento de informações. As DHTs [Stoica et al., 2001, Rowstron and Druschel, 2001, Zhao et al., 2001] utilizam uma função de dispersão (*hash*) para particionar um espaço de chaves entre um conjunto de nós distribuídos, de forma análoga à associação entre chaves e valores de uma tabela de dispersão (*hash table*) tradicional. Ademais, aplicações distribuídas que fazem uso deste tipo de infra-estrutura herdam aspectos de escalabilidade, robustez e facilidade de operação, inerentes à DHT subjacente.

Estes sistemas de armazenamento em nuvem fornecem uma plataforma escalável para acessar e gerenciar dados, e provaram ser bem sucedidos para aplicações web de larga escala [Chang et al., 2006, Lakshman and Malik, 2010]. Mas descartar alguns dos alicerces dos bancos de dados relacionais pode ser considerado um retrocesso, prejudicando fatores importantes tais como a independência de dados, transações consistentes, e outras características fundamentais muitas vezes exigidas por aplicações que são o sustento da indústria de banco de dados. Todavia, a natureza descentralizada, auto-organizada e distribuída desses novos sistemas baseados em princípios *peer-to-peer* fornece uma boa base para a construção de um SGBDR distribuído.

1.1 Repositório de dados distribuídos (DHT)

DHTs armazenam blocos de dados em centenas ou milhares de máquinas conectadas a uma rede (local ou geograficamente distribuída), replica os dados para fins de confiabilidade, e localiza rapidamente os dados. A DHT aborda problemas que são comuns a diversos sistemas distribuídos, tais como tolerância a falhas e escalabilidade, sem a necessidade de esforço adicional por parte das aplicações que a utilizam. Assim, esses mecanismos são realizados uma única vez, na infra-estrutura, e não como parte das aplicações. Essa abordagem proporciona uma infra-estrutura de armazenamento quase universal devido a sua atuação na implantação de uma variedade de cenários.

A DHT fornece uma interface genérica, que facilita sua adoção como substrato de armazenamento: a operação *put* armazena os dados associados a uma chave no sistema; a operação *get* recupera os dados; a operação *delete* remove os dados. A figura 1.1 exemplifica o uso desta interface. A conveniência desta interface de armazenamento incentivou a adoção de DHTs mesmo em sistemas pequenos, nos quais a capacidade do sistema para encaminhar consultas sem manter informações de localização é desnecessária. DHTs nestes cenários são atraentes porque além de reduzirem o ônus do desenvolvimento da infra-estrutura, também proporcionam um potencial de expansão.

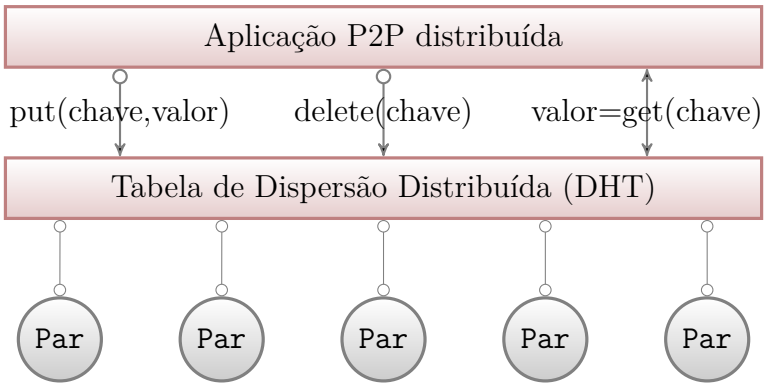

Figura 1.1: Interface de programação para aplicativos baseados em DHT

De maneira formal, uma DHT é uma estrutura de dados descentralizada, que associa chaves a valores e que é distribuída através de vários nós (pares) e é organizada de forma que a entrada, saída ou falha de nós que a compõem tenha um impacto mínimo no seu funcionamento. Essa característica também a torna extremamente escalável para um grande número de nós.

Arquiteturas P2P baseadas em DHT representam uma abordagem atraente para a construção de repositórios de dados (*datastore*) distribuídos, em razão de suas características de descentralização, redundância, adaptabilidade, capacidade de auto-organização e baixo custo operacional. Em resumo, sistemas baseados em DHT fornecem soluções escaláveis e distribuídas de armazenamento de dados com localização eficiente de itens de dados e balanceamento de carga entre os diversos nós da rede.

1.2 Características

A tecnologia P2P pode ajudar a reduzir custos do sistema e permite a partilha de custo através da divisão e uso de infra-estrutura existente e agregação de recursos de locais

diferentes. A agregação de recursos adiciona valor além da mera acumulação de recursos. Por exemplo, sistemas P2P são mais resistentes devido a sua natureza distribuída e não-hierárquica.

Aplicações distribuídas de larga escala são normalmente concebidas com escalabilidade, disponibilidade e robustez em mente, mas uma questão que é freqüentemente esquecida é a simplicidade de implementação e implantação. No entanto, na prática, muitas vezes este é um desafio importante e difícil. Isto pode ser particularmente evidenciado em SGBDRs que são altamente distribuídos em ambientes heterogêneos, onde os dados encontram-se espalhados por diversos locais de armazenamento. Dependendo da forma como a distribuição de dados do SGBDR é realizada, ela pode ser de difícil implantação e altamente subordinada a sistemas de telecomunicações poderosos e confiáveis.

Um Sistema Gerenciador de Banco de Dados Distribuído (SGBDD), que esconde a sua natureza distribuída e paralela do usuário final é mais complexo do que um centralizado. O aumento da complexidade se traduz em uma infra-estrutura cuja aquisição e manutenção requer esforços adicionais para garantir a integridade do banco de dados. Estes fatores resultam em um sistema cujos custos de aquisição e manutenção são mais elevados do que os de um centralizado. O alto custo, somada à necessidade de mão-de-obra especializada para gerir a estrutura distribuída, é uma desvantagem intrínseca. Esses fatores acabam por impactar o custo de desenvolvimento e implantação de um SGBDD.

Uma forma de reduzir a complexidade e custos inerentes a um SGBDR distribuído tradicional é separar a implantação e gerência do banco de dados dos controles de integridade, coordenação e comunicação, visto que a descentralização de funcionalidades propicia o compartilhamento de recursos. Este princípio de decomposição em camadas é comumente usado para simplificar a construção de sistemas complexos. Por exemplo, a rígida separação de camadas entre os protocolos IP e TCP permite que a camada física da rede possa lidar com as operações complexas de entrega de pacotes e que os transmissores e receptores possam lidar com confiabilidade e controle do congestionamento.

Em uma camada superior, tabelas de dispersão distribuídas são freqüentemente citadas como desempenhando um papel semelhante na construção de aplicações descentralizadas. Construir uma aplicação em cima de uma DHT libera projetistas de tratar de questões de escalabilidade e robustez diretamente. Tal abordagem simplifica muito a construção de aplicações distribuídas.

1.3 Objetivo

Este trabalho propõe uma plataforma de dados relacional na nuvem, cujas funcionalidades são concedidas através de uma simbiose entre as características de SGBD tradicionais e a disponibilidade e escalabilidade dos sistemas de armazenamento em nuvem. Esta plataforma pode ser vista como uma área (ou serviço) de armazenamento em rede que é compartilhada entre um conjunto de SGBDRs. Ao passo que a área de armazenamento de dados oferece escalabilidade, replicação de dados e consistência transacional forte, o SGBDR fornece uma linguagem de alto nível para consultar, manipular e definir os dados armazenados na nuvem. O propósito central é demonstrar a viabilidade de fornecer funcionalidades de banco de dados relacional sobre um serviço de armazenamento de dados distribuído sem sacrificar escalabilidade e consistência.

Com base nesta abordagem, é desenvolvido o sistema Phoenix, cuja arquitetura é composta por três camadas: aplicação cliente, banco de dados relacionais e área de armazenamento de dados em nuvem (na forma de *datastore* distribuído). O banco de dados relacional e o *datastore* distribuído são interligados através da decomposição em camadas do sistema. O *datastore*, como camada inferior, propicia um substrato de armazenamento escalável, robusto e eficiente, enquanto que em uma camada superior, o banco de dados relacional permite a criação e manipulação de um modelo lógico da informação armazenada.

A escolha de um *datastore* distribuído para armazenamento dos dados se deve às propriedades desejáveis de durabilidade, disponibilidade, desempenho e descentralização. A recuperação dos dados, que passa a ser um problema de gerência de dados P2P, provê um novo ambiente para o processamento de comandos relacionais, proporcionando ganhos em escalabilidade, confiabilidade e desempenho.

Independência de dados entre os componentes é realizada através de uma interface bem definida, baseada em operações da área de armazenamento de dados, que não assume nem exige qualquer conhecimento das alterações da topologia de roteamento ou o comportamento da rede de armazenamento de dados. Essa independência é um princípio fundamental do projeto do sistema e está intrinsecamente ligada à arquitetura do sistema. De maneira semelhante a SGBDR tradicionais, a independência de dados é alcançada através da estratificação da arquitetura, no qual o nível físico consiste do serviço

de armazenamento distribuído baseado em chave-valor, e o nível lógico consiste de um SGBDR tradicional.

Este é um elemento fundamental de bancos de dados relacionais que reduz os custos e proporciona uma maior flexibilidade para o processamento de dados. Independência lógica dos dados torna possível alterar a estrutura lógica dos dados com um impacto mínimo sobre as aplicações e, portanto, minimiza os custos de manutenção de software. Independência física de dados permite modificações na estrutura de armazenamento físico sem afetar a visão lógica dos dados.

1.4 Panorama da abordagem proposta

Os tradicionais bancos de dados relacionais geralmente armazenam dados na forma de arquivos formados por um conjunto de registros que se assemelham ao modelo lógico de relações. Haja vista que na arquitetura proposta o nível físico consiste em um repositório de dados distribuído, um dos desafios na arquitetura é a definição de mapeamentos entre os modelo de dados da área de armazenamento (chave-valor) e do SGBDR (relacional), e também transformar as operações definidas no nível lógico do banco de dados relacional para operações correspondentes no armazenamento de dados em nuvem.

Essa abordagem é equivalente ao projeto físico do SGBD tradicional, onde o administrador de banco de dados leva em consideração tanto as propriedades do dispositivo de armazenamento físico, a carga de trabalho e consulta para a definição do armazenamento de dados que maximiza o desempenho do sistema. Em um sistema distribuído, projeto físico também envolve determinar os diferentes níveis de granularidade e fragmentação dos dados.

O objetivo desta arquitetura é viabilizar um sistema de banco de dados relacional escalável e de baixo custo de implantação e manutenção. Ou seja, um sistema que não exige pessoal altamente qualificado e especializado para o ajuste de gestão, que é o caso dos sistemas tradicionais de banco de dados distribuídos. O cenário imaginado é um centro de processamento de dados composto por servidores genéricos, onde o sistema pode ser implantado para fornecer um serviço de banco de dados para a nuvem. Ou seja, os dados são armazenados na nuvem, mas podem ser acessados usando a interface padrão do SGBD, e com as mesmas funcionalidades esperada deles. A arquitetura proposta

também é adequada para o armazenamento de grandes volumes de dados dentro de uma única empresa.

A idéia central ao se utilizar um *datastore* não é paralelizar a execução de uma única transação, mas criar um ambiente massivamente distribuído que comporte grande volume de dados, que possa ser distribuído através de milhares de máquinas e que comporte diversas transações tanto de consulta como de inserção em paralelo. O sistema proposto não tem como objetivo substituir SGBDR tradicionais e suas aplicações cujas soluções são centralizadas. O nicho explorado é o de aplicações on-line, interligadas à Internet, que privilegiem robustez, escalabilidade e descentralização, mas sem sacrificar semânticas estritas e exatidão.

1.5 Organização

O restante do trabalho está organizado da seguinte maneira: o capítulo 2 apresenta e compara arquiteturas baseadas em princípios P2P, com enfoque em mecanismos que fomentam plataformas distribuídas de armazenamento. Os capítulos 3 e 4 apresentam, respectivamente, estudos e discussões sobre os modelos de fragmentação e mapeamento de dados do sistema. A descrição do projeto, arquitetura e componentes do sistema é apresentada no capítulo 5. Resultados experimentais são descritos no capítulo 6 e, finalmente, o capítulo 7 delineia as conclusões.

CAPÍTULO 2

ARMAZENAMENTO DE DADOS EM NUVEM

Este capítulo apresenta um resumo da evolução histórica e arquitetônica dos sistemas distribuídos de armazenamento de dados, com foco especial em sistemas que fazem uso de técnicas P2P como substrato para o desenvolvimento de aplicações escaláveis e tolerantes a falhas. A primeira seção apresenta os principais fatos sobre o paradigma P2P e uma breve classificação dos sistemas P2P. As seções posteriores lidam com alguns temas e trabalhos relacionados a este contexto, que abrangem orientações gerais para a construção de sistemas de dados em larga escala. Por fim, as tendências históricas e atuais (por exemplo, serviços em nuvem) são discutidas e contrastadas a fim de distinguir este trabalho das abordagens de outros sistemas.

Do ponto de vista arquitetônico, uma arquitetura centralizada apresenta vários problemas, como um ponto único de falha, escalabilidade limitada em termos de dados e usuários participantes e capacidade reduzida de resistência contra ataques. Em contrapartida, existem argumentos econômicos para a descentralização, tais como a redução de custos e distribuição da carga de gerenciamento de dados e manutenção dos serviços. Fornecer acesso a dados para um grande número de participantes pode ultrapassar rapidamente a capacidades de um fornecedor único e centralizado.

Uma arquitetura distribuída deve apoiar a auto-organização e autonomia, balanceamento de carga, e, preferencialmente, a transparência de localização (usuários acessam os dados sem qualquer conhecimento da localização real dos dados). A escalabilidade precisa ser alcançada em nível de número de participantes e quantidade de dados gerenciados. Além de abordar estes e outros elementos fundamentais, o uso do paradigma P2P vem se mostrando um método eficaz para a concepção de sistemas escaláveis e robustos.

2.1 Paradigma P2P

Peer-to-peer (par a par, ou ponto a ponto) pode ser considerado como um conjunto de conceitos e mecanismos que habilitam a computação distribuída e descentralizada. Refere-se a uma classe de aplicações e sistemas que utilizam recursos computacionais distribuídos

para desempenhar uma função de forma descentralizada. Esse conceito emana de uma arquitetura de rede em que cada nó possui capacidades e responsabilidades equivalentes, diferentemente de arquiteturas cliente/servidor, nas quais alguns nós são dedicados a servir outros.

É necessário salientar que não existe uma definição literal que englobe os diversos tipos de aplicações e sistemas com atributos P2P. Uma definição mais abrangente e amplamente aceita pode ser encontrada em [Androutsellis-Theotokis and Spinellis, 2004]:

> "*Peer-to-peer* são sistemas distribuídos constituídos por nós interconectados com a capacidade de se auto-organizarem em topologias de rede com o objetivo de partilhar recursos, tais como conteúdo, ciclos de CPU, armazenamento e largura de banda, capazes de se adaptar a falhas e acomodar populações de nós transientes, mantendo conectividade e desempenho aceitável, sem que seja necessária a intermediação ou apoio de um servidor global centralizado ou autoridade."

Embora o termo esteja vinculado a uma série de concepções e mecanismos, todos partilham características comuns, tais como a ausência de uma relação hierárquica rígida entre um cliente e um servidor, a autonomia entre nós de (sub)-sistemas vizinhos e o intercâmbio direto de informações entre os nós. Essa forma de organização permite a totalização dos recursos computacionais individuais do sistema e sua partilha entre as entidades participantes.

O modelo *peer-to-peer* introduz uma mudança fundamental em arquitetura de sistemas que o contrasta de duas outras arquiteturas principais: a centralizada e a cliente-servidor. A mudança proeminente que esta arquitetura introduz é a descentralização. Ao contrário das arquiteturas supracitadas que seguem modelos de organização hierarquizados, a arquitetura P2P introduz uma distribuição de controle e responsabilidade. Não existe um único ponto de controle, bem como a responsabilidade pela funcionalidade do sistema é distribuída entre os membros. A Tabela 2.1 apresenta uma comparação entre as características desejáveis em arquiteturas de sistemas [Maly, 2003].

A descentralização consiste em remover qualquer estrutura central da rede de modo que cada ponto possa se comunicar de forma eqüipotente a qualquer outro ponto. Devido a esta estrutura e organização descentralizada, sistemas P2P são inerentemente tolerantes

Característica	Arquitetura		
	Centralizada	Cliente-Servidor	Peer-to-Peer
descentralização	nenhuma	alta	muito alta
conectividade ad-hoc	nenhuma	média	alta
custo de propriedade	muito alto	alto	baixo
anonimato	nenhum	médio	muito alto
escalabilidade	baixa	alta	alta
desempenho	baixo	médio	alto
tolerância a falhas	baixo	médio	alto
auto-organização	média	média	média
transparência	baixa	média	média
segurança	muito alta	alta	baixa
interoperabilidade	padronizada	padronizada	baixa

Tabela 2.1: Comparação das características disponíveis em cada arquitetura.

a falhas. Uma vez que não há ponto central de falha, a perda de um ou mais nós pode ser facilmente compensada. Com a perda de nós o desempenho do sistema pode diminuir, mas deverá continuar operacional desde que haja um número suficiente de nós. No entanto, na dinâmica destes sistemas isso também implica que possa haver uma falta de coerência.

Uma característica importante da descentralização é uma possível implicação, em nível de aplicação, de uma transferência de controle e propriedade dos dados, informações e recursos computacionais para os usuários das aplicações. A forma de concretização da descentralização também é utilizada para classificar os sistemas P2P em duas categorias: sistemas estruturados e não-estruturados, que serão descritos com maiores detalhes nas subseções seguintes (2.1.1, 2.1.2).

Outro mecanismo relevante em sistemas P2P é a capacidade de auto-organização, no sentido de que os diferentes componentes do sistema trabalham em conjunto sem qualquer instância central de coordenação que atribua papéis e tarefas. Neste contexto, o grau de organização em um sistema aumenta sem um aumento de qualquer sistema externo de controle. A estrutura do sistema, bem como sua organização interna, é idealmente construída sem controle ou influência externa.

2.1.1 Sistemas P2P não-estruturados

Em sistemas P2P não-estruturados, não há qualquer protocolo global para posicionamento dos dados e a topologia da rede não é rigorosamente controlada. A estratégia básica de pesquisa é inundar parcialmente a rede P2P com mensagens de busca ou percorrer

aleatoriamente a rede até que os dados desejados sejam encontrados.

Embora técnicas baseadas em inundação sejam resistentes a populações transientes de nós e eficazes para localizar itens altamente replicados, elas são pouco adequadas para a localização de itens raros.

Sistemas P2P não-estruturados oferecem uma maior facilidade de implantação, flexibilidade e menor custo de manutenção do que sistemas estruturados. No entanto, geralmente possuem baixo desempenho de pesquisa e não escalam muito bem, porque a carga de consulta em cada nó cresce linearmente com o número total de consultas, que, por sua vez, cresce com o número de nós no sistema.

2.1.2 Sistemas P2P estruturados

Em sistemas P2P estruturados, a localização dos dados é determinada por um esquema global, como uma função de dispersão (*hash*) que mapeia uma chave de busca a um nó. Assim, as chaves são distribuídas aos nós, formando uma *Distributed Hash Table* (DHT) virtual, sendo cada nó responsável por um sub-conjunto de pares chave-valor. Esta abstração baseada em um esquema de particionamento do espaço de chaves fornece naturalmente um modelo de dados chave-valor.

Sistemas P2P estruturados oferecem maior escalabilidade e disponibilidade, em comparação com sistemas P2P não estruturados e arquiteturas cliente/servidor tradicionais. Limitando a tabela de roteamento de cada nó a um número pequeno de vizinhos, sistemas P2P estruturados formam redes de sobreposição (*overlay*) na qual tempos de busca e manutenção da DHT necessitam apenas de um número de saltos de ordem logarítmica entre os pares que compõem a rede. Além disso, uma estratégia de posicionamento e distribuição de dados baseado em DHT naturalmente leva a um balanceamento de carga no sistema.

Uma das grandes limitações de sistemas P2P estruturados é a busca exata de informações. A fim de se localizar o nó que armazena um item de dado, é preciso saber exatamente seu identificador (chave). Para resolver este problema, é possível conceber técnicas de buscas por palavra-chave sobre consultas de busca exatas. Técnicas que implementam este tipo de abordagem tem sido objeto de intensa pesquisa [Ramabhadran et al., 2004, Zheng et al., 2006].

2.1.3 Modelo de dados

O modelo de dados chave-valor pode ser compreendido como um modelo para a manipulação de dados semi-estruturados. Nos modelos semi-estruturados os dados não estão dispostos de acordo com uma estrutura formal de tabelas e modelos de dados, mas ainda assim contêm termos ou marcadores para separar elementos semânticos e hierarquias de registros e campos dentro dos dados.

Conseqüentemente, bancos de dados chave-valor permitem o armazenamento de dados sem esquemas ou estruturas rígidas definidas. A informação que normalmente é associada com um esquema está contida dentro dos dados, que usualmente é chamada de "auto descritiva". Não existe uma separação clara entre os dados e o esquema, e o grau em que está estruturado depende da aplicação. Em algumas formas de dados semi-estruturados, não há nenhum esquema separado.

No modelo chave-valor, dados e meta-dados são identificados por uma chave. Tal identificador é associado com um valor, formando um par (chave, valor). Bancos de dados chave-valor somente oferecem acesso aos objetos dadas suas chaves. Consultas mais complexas são difíceis de serem processadas já que os critérios de distribuição são baseados em chaves carentes de semântica. Os dados em si normalmente são algum tipo primitivo (binário) ou um objeto que está sendo empacotado. Isso substitui a necessidade de um modelo de dados fixo e torna menos estrita a exigência de dados corretamente formatados.

O modelo chave-valor favorece a simplicidade na estrutura lógica e física dos dados em troca de complexidade nos meta-dados, que, entre outras coisas, assume o papel de representar restrições e integridade referencial. Assim, o modelo oferece um ambiente flexível para gerenciamento de uma base de dados através do fornecimento de uma forma genérica de armazenar informações na forma de objetos, e um método flexível, baseado em atributos, de criar relações entre esses objetos.

2.2 Arquiteturas baseadas em DHT

Arquiteturas baseadas em princípios P2P vêm sendo amplamente aplicadas no cenário atual para fomentar os mecanismos subjacentes das plataformas de armazenamento em nuvem, produzindo abundante pesquisa na área de sistemas distribuídos com foco especial

em bancos de dados semi-estruturados altamente escaláveis e tolerante a falhas [Cattell, 2010]. Essa classe de aplicações vem sendo utilizada para alavancar diversas tecnologias existentes. Por exemplo, em [Huebsch et al., 2005] a DHT é utilizada como um mecanismo para consultas distribuídas sobre um banco de dados relacional; em [Sit et al., 2004] é introduzido um sistema no qual um conjunto de nós colaboram para armazenar cópias de artigos *Usenet* de forma compartilhada e distribuída; em [Walfish et al., 2004a, Walfish et al., 2004b] é proposto um sistema para desemaranhar o serviço de resolução de nomes (DNS) baseado em uma DHT.

Apesar de alguns desses sistemas utilizarem a DHT como o alicerce para diversas abordagens no tratamento de informações e não somente para buscas baseadas em chave, nem sempre suas implementações são decomponíveis da DHT subjacente. Embora essa seja uma alternativa viável para sistemas que utilizem a DHT de forma simples e direta, o mesmo não se aplica a um sistema distribuído complexo composto por grande número de componentes em vários níveis de abstração. Uma abordagem baseada em camadas decomponíveis simplifica muito o projeto e construção de aplicações distribuídas.

Um estudo precursor deste conceito é apresentado em [Chawathe et al., 2005], no qual o sistema mantém estrita utilização de uma abordagem em camadas através da construção inteiramente sobre o serviço OpenDHT [Rhea et al., 2005]. No sistema apresentado, a DHT é utilizada como alicerce para a implementação de uma "infra-estrutura de mapeamento" que possibilita a dissociação da implantação e gestão do sistema daquela da DHT subjacente. Também é analisada a viabilidade da abordagem em camadas para simplificar a construção de aplicações distribuídas e o impacto no desempenho do uso de uma DHT genérica (independente da aplicação).

Em [Huebsch et al., 2005] é apresentado um estudo bastante similar ao aqui abordado e que detalha uma arquitetura chamada PIER (*Peer-to-Peer Information Exchange and Retrieval*). Esse sistema utiliza uma DHT como substrato subjacente de comunicação e fornece uma linguagem de consulta, análoga a de banco de dados relacionais, que fornece transparência de localização e escalabilidade com semânticas mais relaxadas. Na arquitetura PIER são apresentadas técnicas para implementação de consultas de associação (com enfoque em operações de totalização e agregação).

Arquiteturas baseadas em DHT podem ser divididas em duas categorias: as customizadas, que são intrinsecamente dependentes da rede de sobreposição subjacente (DHT)

[Aspnes and Shah, 2007, Harvey et al., 2003b, Bharambe et al., 2004, Huebsch et al., 2005] e as estratificadas, que são dispostas sobre uma DHT existente e não precisam modificar o comportamento ou topologia da rede subjacente [Ramabhadran et al., 2004, Zheng et al., 2006, Gao and Steenkiste, 2004]. Abordagens estratificadas geralmente apresentam bom desempenho de balanceamento de carga, mas exigem algoritmos não triviais de apoio a consultas por abrangência devido ao mecanismo de busca exata da DHT. Já que este trabalho foca no uso de um serviço DHT como um alicerce, serão abordadas apenas propostas que não assumam conhecimento nem exijam alterações à topologia da DHT subjacente e que ofereçam suporte nativo a consultas por abrangência.

2.3 Repositório de dados distribuídos

Bancos de dados relacionais são concebidos a partir de modelos onde uma coleção de dados é representada em termos de entidades e relacionamentos, mas esse modelo não propicia uma escalabilidade horizontal[4] para aplicações cliente. A busca por uma solução deste problema resultou num redesenho arquitetônico de sistemas gerenciadores de dados, redirecionando-os a uma gama de aplicações focadas em requisitos de alta escalabilidade, disponibilidade e baixa latência.

Em um artigo recente [Cattell, 2010], esses sistemas foram classificados em três grupos, de acordo com seus modelos de dados: bancos de dados chave-valor, como o Scalaris [Berlin and onScale solutions GmbH, 2010]; bancos de dados orientados a documento, como o SimpleDB [Amazon, 2007]; e bancos de dados tabulares, como o BigTable [Chang et al., 2006], que é o banco de dados subjacente usado pelo Google App Engine [Google, 2008]. Nessa classificação, bancos de dados chave-valor são baseadas em valores e um índice para encontrá-los a partir de uma chave definida pelo usuário, bancos de dados orientados a documento definem uma chave para um conjunto de pares atributo-valor, enquanto que bancos de dados tabulares seguem um modelo de fragmentação orientado a colunas.

Em nível básico, o objetivo destes sistemas é o de sustentar, com desempenho e disponibilidade adequados, um serviço de consulta e armazenamento sobre um grande conjunto de dados, sem excesso significativo de provisionamento. O requisito de utilização dos recursos exige que o sistema seja altamente dinâmico. Embora cada um tenha caracte-

[4] *scale-out*, aumentar a capacidade e redistribuir a carga de dados e processamento ao acrescentar novos elementos a um agrupamento de servidores.

rísticas únicas, esses sistemas focam na harmonização de aspectos como disponibilidade, desempenho, escalabilidade, consistência e custo.

Funcionalidades complexas de relacionamentos e processamento, comuns em SGBDR tradicionais, são invariavelmente implementadas pelas aplicações cliente externas ao sistema. Essa estratégia se adapta bem a um amplo cenário de aplicações web, mas se torna menos atraente para as necessidades de empresas comuns, nas quais não há uma grande equipe de desenvolvimento para customização e manutenção da aplicação.

Nas próximas seções, são apresentados detalhes sobre alguns destes sistemas.

2.3.1 SimpleDB

SimpleDB [Amazon, 2007] é um serviço web ofertado através da plataforma de serviços da Amazon para a execução de consultas sobre dados estruturados em tempo real. O componente central deste serviço é um repositório de dados distribuído orientado à atributos chave/valor, construído a partir do sistema apresentado em [DeCandia et al., 2007].

Uma característica fundamental deste sistema é a utilização de consistência eventual, ou seja, alterações podem não ser vistas por operações de leitura subseqüentes. Embora na maioria das vezes essa situação não ocorra, a possibilidade deve ser levada em conta se houver requisitos de coerência de dados.

2.3.1.1 Modelo de dados

O modelo de dados do SimpleDB representada os dados de forma semi-estruturada e sem esquema pré-definido (*schemaless*). Apesar de não seguir um modelo relacional, existe uma certa correspondência de terminologia: o banco de dados é representado por uma coleção de domínios (tabelas ou relações) que armazenam coleções de itens. Itens (linhas ou tuplas) são coleções de pares atributo/valor. Atributos são qualidades (colunas ou propriedades) dos itens e correspondem a colunas esparsas. Atributos podem ser compostos por múltiplos valores. Todos os valores são indexados à medida que são adicionados.

Conforme esboçado na figura 2.1, um domínio corresponde de certa forma a uma tabela. São permitidos até 100 domínios e todas as consultas devem ser realizadas por domínio. Dentro de um domínio, é possível criar itens e cada item deve possuir um identificador presumivelmente único. Cada item tem atributos e atributos têm valores.

Nome	Idade	Sexo	Cidade	Estado	CEP	Produtos
Liza Smith	25	Feminino	Rio Branco	Acre	40210-104	Livros
John Smith	47	Masculino	Salvador	Bahia	96003-035	Revistas

Domínio Items Atributos

Figura 2.1: Modelo de dados do SimpleDB

Não é preciso (nem possível) declarar o esquema de um item e qualquer item pode ter até 256 atributos.

Há também diversas limitações quanto ao formato e tamanho dos itens e valores de dados que podem ser armazenados no SimpleDB. Todos os valores devem ser do tipo *string* (cadeia de caracteres alfanuméricos) e de no máximo 1024 *bytes* de comprimento, limitando a quantidade de texto que pode ser armazenado em um único atributo. Mas esse limite pode ser contornado através da adição em forma encadeada de até 256 atributos por item.

2.3.1.2 Linguagem de pesquisa

O SimpleDB oferece uma linguagem de consulta simples e personalizada, apresentada na figura 2.2, que introduz um comando similar à cláusula *SELECT* da linguagem de consulta estruturada SQL para banco de dados relacionais. Esse comando permite consultas sobre pares de atributo/valor associados a itens.

```
SELECT <saída> FROM domínio  [WHERE condição]
                             [ORDER BY atributo {ASC | DESC}]
                             [LIMIT contador]

   <saída> ::=  * | COUNT(*) | (atributo [, atributo])
<condição> ::=  condição [ AND | OR ] condição
<condição> ::=  atributo { < | <= | > | >= | = | != } atributo
```

Figura 2.2: Linguagem de pesquisa do SimpleDB

Predicados são aplicados sobre atributos e cada valor do atributo é considerado individualmente sobre a condição de comparação definida no predicado. Nomes de itens são selecionados se um dos valores do atributo satisfizer a condição do predicado.

A linguagem também dá suporte aos operadores de união e interseção da teoria dos conjuntos, mas existe uma diferença importante entre operadores de comparação dentro de um predicado e operações de conjunto entre predicados. Operadores de comparação são aplicados sobre o conjunto de valores de um único atributo, enquanto que operadores de conjuntos são aplicados sobre os nomes dos itens resultantes de cada predicado (os valores dos atributos não são relevantes). Os operadores de união e interseção somente retornam nomes de itens presentes em um ou ambos os conjuntos de resultados, respectivamente.

Operações de ordenação de dados são restritas a um único atributo e por ordem ascendente (padrão) ou decrescente. Todas as operações são realizadas em ordem lexicográfica.

2.3.2 App Engine

A plataforma para aplicações web App Engine [Google, 2008] do Google também conta com um arcabouço de armazenamento e manipulação de dados semi-estruturados e sem esquema chamado de App Engine Datastore. Projetado para aplicações web, com ênfase no desempenho de operações de leitura e consulta, o sistema também dá suporte a transações atômicas e coerência.

Esse repositório de dados pode ser considerado como uma interface simplificada sobre o BigTable [Chang et al., 2006], um banco de dados orientado a colunas e modelado na forma de um dicionário multidimensional, esparso, ordenado e persistente. No entanto, ao contrário do SimpleDB, atualmente a interface não é acessível por aplicações externas à plataforma de serviços web do Google.

2.3.2.1 Modelo de dados

A unidade básica de armazenamento de informação do App Engine Datastore é uma entidade, que é composta de uma ou mais propriedades. Cada entidade é identificada por uma chave única na forma de um número inteiro ou *string* cujo valor pode ser fornecido pelo aplicativo ou de forma automática pelo serviço de armazenamento de dados.

As propriedades de uma entidade são representadas por um mapa nome-valor. O nome de cada propriedade é único e de tipo *string*, mas cada propriedade pode ter um ou mais valores de tipos mistos, incluindo números inteiros, valores de ponto flutuante, *strings*, datas, dados binários, entre outros.

Chave	Atributo	Valor
1	Nome	Liza Smith
1	Idade	25
1	Sexo	Feminino
2	Nome	John Smith
2	Idade	47
2	Sexo	Masculino

Figura 2.3: Modelo de dados do Google App Engine

Entidades formam hierarquias distintas e estritas dentro do repositório de dados, sendo que cada entidade pode ter uma entidade ancestral. Uma entidade raiz (entidade sem ancestral) e seus descendentes formam um grupo de entidades conhecidas como um *grupo de entidade*. A relação ancestral entre entidades e seus correspondentes grupos de entidades desempenham um papel central na forma que os dados são armazenados e operados (transações).

2.3.2.2 Linguagem de pesquisa

A recuperação de entidades de dados armazenadas no App Engine pode ser realizada através da linguagem GQL (*Google Query Language*). A sintaxe da linguagem, apresentada na figura 2.4, é semelhante a SQL, mas não oferece os mesmos recursos, limitando-se ao comando *SELECT*. Uma consulta opera em toda entidade de um determinado tipo (uma classe de dados) e recupera entidades do armazenamento de dados que atendam a um conjunto de condições.

Uma consulta especifica um tipo de entidade, zero ou mais condições com base em valores de propriedades da entidade (às vezes denominados “filtros”) e zero ou mais descrições de ordem de classificação. Quando a consulta é executada, ela obtém todas as entidades de um determinado tipo que atendam a todas as condições dadas, classificadas na ordem descrita.

Para obter uma entidade do armazenamento de dados, um aplicativo pode utilizar a sua chave ou realizar uma consulta correspondente às propriedades da entidade. Uma consulta pode retornar zero ou mais entidades e os resultados classificados por valores de propriedade. Uma consulta também pode limitar o número de resultados retornados pelo

```
SELECT * FROM <tipo>  [WHERE <condição> [AND <condição> ...]]
                      [ORDER BY <propriedade> [ASC | DESC]
                      [, <propriedade> [ASC | DESC] ...]]
                      [LIMIT [<offset>,]<count>] [OFFSET <offset>]

<condição> ::=  <propriedade> { < | <= | > | >= | = | != } <valor>
<condição> ::=  <propriedade> IN <lista>
<condição> ::=  ANCESTOR IS [ <propriedade> | <chave> ]
```

Figura 2.4: GQL: Linguagem para a recuperação de entidades de dados

armazenamento de dados, para economizar memória e tempo de execução.

O mecanismo de consulta impõe algumas restrições quanto às condições de filtragem e classificação. Para filtrar ou classificar uma propriedade, é necessário que a mesma exista e possua um valor. Os filtros de desigualdade (<, <=, >=, >, !=) podem ser utilizados uma ou mais vezes em uma consulta, mas somente em uma única propriedade, ou seja, não é permitido utilizar filtros de desigualdade em duas ou mais propriedades diferentes na mesma consulta.

2.3.2.3 Transações

Transações são executadas através da rede distribuída usando um *grupo de entidade*. Uma transação manipula entidades dentro de um único grupo. Entidades do mesmo grupo são armazenadas juntas para a execução eficiente das operações. Entidades podem ser atribuídas a grupos quando criadas.

Criar, atualizar ou excluir uma entidade ocorre em uma transação. Uma transação garante que toda alteração feita na entidade seja salva no armazenamento de dados, ou, no caso de uma falha, nenhuma das alterações seja feita. Isso garante a consistência dos dados em uma entidade.

É possível fazer alterações em diversas entidades dentro de uma única transação. Mas é preciso informar com antecedência quais entidades serão atualizadas. Ao criar uma entidade, é preciso declarar que uma entidade pertence ao mesmo grupo de entidades que outra entidade. Todas as entidades obtidas, criadas, atualizadas ou excluídas em uma transação devem obrigatoriamente estar no mesmo grupo de entidades.

O armazenamento de dados usa controle de concorrência otimista para gerenciar transações. Antes de uma instância do aplicativo aplicar alterações num grupo de entidades, a transação verifica se outras transações modificaram os dados a serem atualizados. Se a verificação revelar alterações em conflito, a transação falha instantaneamente. O aplicativo pode tentar a transação novamente para aplicá-la aos dados atualizados.

2.3.3 Scalaris

Scalaris [Schütt et al., 2008, Berlin and onScale solutions GmbH, 2010] é um banco de dados chave-valor (*key/value datastore*) que utiliza uma rede de sobreposição para alcançar disponibilidade através da replicação de dados e transações distribuídas para manutenção da coerência dos dados.

O sistema é concebido em três camadas (comunicação, replicação e transação) que em conjunto garantem propriedades ACID (Atomicidade, Consistência, Isolamento, Durabilidade) sobre uma rede de sobreposição, proporcionando um banco de dados chave-valor baseado em DHT que dá suporte a leituras e escritas monotônicas, garantindo que após a leitura inicial de um objeto por um dado processo, leituras sucessivas sempre retornarão o mesmo valor (ou mais recente dependendo do contexto), enquanto que uma operação de escrita num objeto é aplicada em todas as réplicas antes de qualquer escrita sucessiva pelo mesmo processo neste objeto.

A camada inferior, ou camada de comunicação, é composta por uma rede de sobreposição estruturada com desempenho logarítmico em buscas e serve como substrato para a construção de uma estrutura de dados chave-valor multidimensional. Esta rede de sobreposição é implementada utilizando-se o protocolo de endereçamento Chord# [Schütt et al., 2006, Schütt et al., 2007] que armazena chaves em ordem lexicográfica, possibilitando consultas por abrangência.

A camada intermediária, ou camada de replicação e transação, provê propriedades ACID em face de operações concorrentes de escrita através de uma variação do protocolo de consenso *Paxos Commit* [Gray and Lamport, 2006, Haridi and Moser, 2007] integrado à rede de sobreposição. O protocolo é utilizado para implementar transações sobre múltiplias chaves e também assegura que todas as réplicas de uma chave são atualizadas de forma consistente.

A camada superior, ou camada de aplicação, exporta a interface do banco de dados

chave-valor, disponibilizando um serviço de armazenamento escalável e tolerante a falhas. Essa camada pode ser utilizada como mecanismo subjacente de diversos outros tipos de sistema, tais como aplicações on-line.

Em suma, os mecanismos de replicação e transação, desenvolvidos sobre a rede virtual, garantem propriedades ACID em face de acesso concorrente aos dados. A disponibilidade é alcançada através de replicação simétrica [Ghodsi et al., 2005] e consistência é fornecida através de uma variação do protocolo de consenso Paxos. O protocolo de consenso também é utilizado para implementar transações que envolvam várias chaves e para garantir que todas as réplicas de uma chave sejam atualizadas de forma consistente. A interface para o armazenamento de chave-valor escalável e tolerante a falhas consiste em operações de leitura, escrita e remoção de itens de dados baseado em chaves, como em outros sistemas baseados em DHT.

2.4 Discussão

Nesta seção serão abordados e contrastados dois aspectos que mais caracterizam os serviços de armazenamento em nuvem: o modelo de dados e consistência. Por fim, é apresentado uma comparação entre as abordagens.

2.4.1 Modelo de Dados

Uma característica marcante entre os sistemas de armazenamento em nuvem é uma inclinação a modelos de dados auto-descritivos (semi-estruturados). Por exemplo, os modelos de dados disponibilizados pelos sistemas SimpleDB e App Engine se assemelham ao modelo Entidade-Atributo-Valor (EAV) [W. W. Stead and Straube, 1983], uma representação do conhecimento de uso geral que surgiu com o conceito de listas de associação (pares de atributo-valor). Por outro lado, pode-se dizer que o Scalaris segue um modelo BLOB (*Binary Large OBjects*, ou grandes objetos binários), que é utilizado para representar tipos de dados não identificados.

O modelo BLOB permite uma maior flexibilidade e pode ser considerado como uma porção de memória na qual uma determinada quantidade de dados, em forma binária, pode ser escrita. Sistemas que seguem esse modelo não guardam informações sobre os relacionamentos, significado ou utilidade dos dados, que fica a cargo das aplicações clien-

tes. Diferentemente do modelo BLOB, o modelo semi-estruturado define uma estrutura conceitual em que os dados geralmente são representados em três colunas: entidade, atributo e valor. Em um banco de dados semi-estruturado, o esquema lógico é diferente do esquema físico, enquanto que num banco de dados convencional ambos são similares. Um sistema que lida com dados semi-estruturados deve ter meios de traduzir as características físicas do esquema em um esquema lógico que reflita a estrutura e relação dos dados. Isto é conseguido através de metadados cujo conteúdo define a semântica do domínio a ser modelado.

No SimpleDB e App Engine, consultas podem ser realizadas sobre os dados semi-estruturados através de linguagens próprias de consulta cuja sintaxe é semelhante ao SQL. A linguagem de consulta do SimpleDB é limitada à cláusula SELECT, que pode ser usada para pesquisa sobre pares de nome-valor associados a itens. A linguagem possui diversas limitações: os operadores de ordenação podem ser aplicados apenas a um único atributo, operações com conjuntos são restritas aos nomes de itens, e não é possível executar consultas entre domínios diferentes. A linguagem de consulta do Google App Engine, conhecida como GQL, também é limitada à cláusula SELECT, em que um ou mais predicados podem ser definidos sobre atributos.

Esses serviços de armazenamento em nuvem, principalmente os orientados a modelos chave-valor e semi-estruturado, não oferecem independência lógica dos dados. Mesmo que alguns possuam uma linguagem de consulta similar ao SQL, que permite a recuperação de dados a partir de chaves, os resultados das consultas são compostas por representações "empacotadas" de todos os atributos associados a uma determinada chave. Assim, a responsabilidade de interpretar o valor retornado a fim de determinar os valores individuais de cada atributo é delegada à aplicação cliente. Esta é, em parte, a razão por que nestes sistemas não há distinção clara entre os dados e esquema e, portanto, a semântica do domínio sendo modelado é escondida dentro das aplicações. Esse excesso de simplificação no modelo de dados afeta significativamente a complexidade das aplicações e, eventualmente, leva a um aumento nos custos de manutenção.

Embora a escolha do modelo de dados dependa do domínio e exigências de flexibilidade, as necessidades que levaram sistemas baseados em banco de dados hierárquicos e arquivos planos (*flat files*) a migrar para bancos de dados relacionais também eventualmente levarão esses sistemas baseados em modelos semi-estruturados a evoluir de forma semelhante.

Apesar de o modelo ser eficiente para consultas dentro de uma mesma entidade, a falta de um esquema formal de metadados acaba por aumentar significativamente a complexidade da coleta e visualização de informações a partir dos dados.

2.4.2 Concorrência e consistência de dados

Apesar de ambos os sistemas, SimpleDB e App Engine, oferecerem suporte a transações e consistência, este é um ponto onde há uma diferença fundamental. O SimpleDB, bem como a maioria dos serviços de armazenamento distribuído, somente dá suporte à consistência eventual, que, sob certas circunstâncias, significa que uma operação de escrita já concluída pode não ser vista por operações de leitura subseqüentes. Esta noção de consistência relaxada é conhecida como BASE (*Basically Available, Soft state, Eventually consistent*).

Por outro lado, as transações do Google App Engine garantem propriedades ACID. As operações de leitura dentro de uma transação operam sobre uma única visão instantânea e consistente dos dados (incluindo escritas dentro da transação). Um aplicativo pode executar múltiplas operações de inserção, atualização ou remoção de itens de dados em uma única transação. No entanto, em uma única transação somente é possível operar em itens de dados pertencentes a um mesmo grupo. Na plataforma App Engine, um grupo é composto por um conjunto arbitrário de elementos, definidos pelo usuário, que devem ser armazenados num mesmo nó, ou em nós próximos. O grupo define uma unidade básica de consistência, escalabilidade e replicação. Embora o App Engine forneça consistência forte e alta disponibilidade, os itens de dados em si não são explicitamente replicados, uma vez que o o BigTable faz uso, de forma transparente, do sistema de arquivos distribuído *Google File System* [Ghemawat et al., 2003] para garantir confiabilidade e disponibilidade.

Coerência, disponibilidade e tolerância a partições da rede (CAP, sigla do inglês *Consistency, Availability, Partition Tolerance*) são propriedades desejáveis em qualquer sistema distribuído. No entanto, de acordo com o teorema CAP [Brewer, 2000], somente é possível assegurar, no máximo, duas dessas propriedades em qualquer sistema distribuído com dados compartilhados. Assim, a maioria dos sistemas de armazenamento em nuvem assegura disponibilidade e tolerância a partições na rede, em detrimento à consistência forte. Eles geralmente dão suporte a uma noção fraca de consistência e, portanto, é possível que existam duas cópias de um mesmo item em desacordo sobre seus valores. Embora

isto possa não ser um grande problema para as aplicações OLAP, consistência forte é um importante requisito para aplicações OLTP. Scalaris [Schütt et al., 2008] e G-Store [Das et al., 2010] são exemplos de sistemas que têm sido propostos para resolver o problema de consistência, os quais favorecem consistência em detrimento à disponibilidade ou tolerância a partições na rede.

À luz do teorema do CAP, o Scalaris prioriza consistência rígida e tolerância a partições na rede, sacrificando a disponibilidade, para assegurar que os dados estejam sempre consistentes. Em caso de partições na rede, as operações somente serão bem sucedidas na partição que engloba a maioria das réplicas de um determinado item. Portanto, algoritmos baseados em quorum são usados para garantir consistência mediante operações em grande maioria. Um aspecto desejável do suporte a transações do Scalaris é a possibilidade de operar sobre dados arbitrários, enquanto que na App Engine é preciso informar com antecedência quais entidades participarão da transação e as entidades devem pertencer ao mesmo grupo.

2.4.3 Comparação das abordagens

Projetados para aplicações web, com ênfase no desempenho das operações de leitura e consultas, os serviços de armazenamento em nuvem fornecem suporte limitado para aplicações OLTP comuns. Isso ocorre porque a maioria deles não suporta consistência forte e somente fornecem uma linguagem de consulta simples que não permite uniões (álgebra relacional). Apesar de ser possível projetar uma linguagem de consulta com bom desempenho para determinados tipos de consultas, como aquelas que envolvam uma única relação ou domínio, manipulações de dados que não podem ser expressas na linguagem devem ser codificadas dentro da aplicação. Isso resulta em trabalho extra para o desenvolvimento de novas aplicações. Além disso, os aspectos de consistência e de linguagem de consulta estão intimamente relacionados ao particionamento físico e modelo de distribuição dos dados, contradizendo noções tradicionais de independência lógica e física. Um exemplo desta estreita interligação é o conceito de entidades (grupos) adotado pela App Engine. Em suma, é possível concluir que o objetivo desses sistemas é apoiar, com bom desempenho e disponibilidade, um serviço de armazenamento para grandes volumes de dados, sem incorrer em complexidade excessiva. Eles estão em algum lugar entre a funcionalidade fornecida por arquivos planos e SGBDRs para sistemas centralizados, enquanto

que o objetivo do sistema proposto neste trabalho é o desenvolvimento de um banco de dados escalável e completo cujos dados são armazenados na nuvem.

Apesar destes sistemas de armazenamento oferecerem um quadro evolutivo para gerenciamento de dados e serem bem sucedidos para aplicações webs de larga escala [Chang et al., 2006, Lakshman and Malik, 2010], descartar alguns dos fundamentos de bancos de dados relacionais pode ser considerado um enorme retrocesso, já que prejudica fatores importantes como a independência de dados, transações consistentes, e outras características muitas vezes exigidas por aplicativos que são fundamentais para a indústria de banco de dados. Em particular, a maioria dos atuais sistemas de armazenamento em nuvem são adequados para aplicações OLAP, mas não para OLTP [Abadi, 2009], principalmente porque é difícil manter consistência forte entre réplicas que podem ser distribuídas por grandes distâncias geográficas.

Agrupamento (*cluster*) de bancos de dados [Röhm et al., 2000, Röhm et al., 2001, Röhm et al., 2002, Gançarski et al., 2002] também têm sido proposto para fornecer armazenamento de dados relacionais em larga escala. Nestes sistemas, uma camada é adicionada no topo de um *cluster* de SGBDRs, cada um deles rodando em máquinas genéricas. Esta camada é responsável por armazenar metadados e gerenciar a execução de consultas e replicação. A abordagem proposta neste trabalho difere desta arquitetura ao manter a mesma abordagem em três camadas de um SGBDR tradicional, mas com um sistema de armazenamento em larga escala como seu componente físico. Ou seja, em bancos de dados *cluster*, a camada superior fornece uma visão única dos dados processados e armazenados em um conjunto de SGBDR tradicionais. No sistema proposto neste trabalho, por outro lado, existe um único armazenamento de dados distribuídos, em cima do qual um conjunto de SGBDR tradicionais pode ser executado. Embora a maioria dos bancos de dados de *cluster* ter sido projetado para paralelização de consultas OLAP, nosso objetivo principal não é paralelizar a execução de uma única transação, mas criar um ambiente maciçamente distribuído que seja potencialmente capaz de armazenar grandes quantidades de dados, ser distribuído entre milhares de máquinas e que possa processar eficientemente um grande número de transações simultâneas.

A arquitetura e os objetivos do sistema proposto neste trabalho são semelhantes aos do MySQL Cluster [Ronström and Thalmann, 2004]. Eles são similares porque são baseados em um mecanismo de armazenamento (NDB) que maneja dados armazenados em

um conjunto de nós. As abordagens divergem sobre como a disponibilidade do sistema é assegurada através da replicação de dados. O MySQL Cluster utilizada um modelo de replicação mestre/escravo (*master/slave*) baseado em registros (*logs*), enquanto que este trabalho herda o modelo de replicação do serviço de armazenamento subjacente baseado em DHT. De acordo com [Cattell, 2010], o MySQL Cluster sofre com pontos de estrangulamento após uma dúzia de nós. O projeto do sistema proposto neste trabalho pode, potencialmente, ter uma melhor escalabilidade, dado que é baseado em uma tecnologia que tem se mostrado capaz de gerenciar grandes quantidades de dados.

Dentre os sistemas de armazenamento distribuído, pode ser citado o OceanStore [Kubiatowicz et al., 2000], que é um projeto acadêmico de pesquisa em andamento na Universidade de Berkeley, Califórnia. O OceanStore fornece uma arquitetura de escala global para armazenamento persistente. Como os trabalhos mencionados anteriormente, o OceanStore segue um modelo de infra-estrutura de utilidade. O projeto foca em aspectos gerais da gestão de dados em larga escala, bem como segurança e privacidade. Não é discutido em detalhe o processamento de consultas de banco de dados. Ainda, nesta linha de pesquisa, é relevante mencionar o PNUTS [Cooper et al., 2008]. O PNUTS, desenvolvido pelo Yahoo, é uma plataforma de banco de dados distribuído destinada a aplicações Web. PNUTS destina-se a um serviço de armazenamento simplificado de banco de dados para conjuntos de dados extremamente grandes. Ele usa uma DHT como sua camada física.

Cobrir a lacuna entre os modernos sistemas de armazenamento em nuvem e sistemas de banco de dados tradicionais também tem sido o foco de outros trabalhos recentes [Curino et al., 2010, Egger, 2009]. A integração entre MySQL e Cassandra [Lakshman and Malik, 2010] é descrito em [Egger, 2009]. Embora a arquitetura do sistema seja semelhante à abordagem deste trabalho, algumas questões importantes para a integração do *datastore* com a tecnologia de banco de dados relacional, tais como mapeamento de dados e funcionalidades esperadas do serviço de armazenamento (*datastore*) subjacente não são analisadas. O sistema descrito em [Curino et al., 2010] tem objetivos semelhantes, mas foi desenvolvido com uma arquitetura diferente, que consiste de um *cluster* de servidores executando SGBDR tradicionais e com um mecanismo de transação distribuída sobre o *cluster*.

No próximo capítulo são abordadas questões de integração e distribuição de dados que são relevantes para a utilização da DHT como substrato de armazenamento.

CAPÍTULO 3

INTEGRAÇÃO E DISTRIBUIÇÃO DE DADOS

Este capítulo aborda dois aspectos centrais para o armazenamento de dados relacionais estruturados em uma tabela de dispersão distribuída: como integrar os diferentes modelos de dados e como distribuir os dados na rede P2P. Essas questões impactam como dividir as relações, em quais atributos devem ser aplicadas as funções de mapeamento e dispersão, e se há benefício em mapear, fragmentar ou agrupar determinadas relações.

A idéia de fragmentar dados e de distribuí-los entre diversos nós não é nova, sendo tema recorrente da área de bancos de dados distribuídos. A principal diferença entre os sistemas de banco de dados distribuído e sistemas P2P é a autonomia dos nós que compõem o sistema. Os nós de um banco de dados distribuído estão sob o controle de um sistema central de gerenciamento de banco de dados, que organiza os dispositivos de armazenamento localizados num mesmo local físico ou distribuídos em uma rede de computadores, provendo assim uma visão global (do banco de dados) expressa em um modelo de dados comum. Sob o controle de um coordenador central, os dados armazenados no banco de dados são fragmentados e distribuídos entre todos os locais de armazenamento. A cada nó do banco de dados é atribuída uma partição dos dados pelos quais o nó é responsável. De forma similar a sistemas P2P, os dados podem ser replicados para aumentar a confiabilidade e tolerância a falhas, dependendo das necessidades de negócio. A distribuição dos dados é transparente para os usuários, ou seja, um usuário não precisa saber onde os dados são armazenados ou recuperados.

Sistemas P2P aplicam o mesmo conceito de fragmentação e partilha de dados que em bancos de dados distribuídos, mas sem controle central e com nós autônomos. No entanto, os dados devem ser organizados de maneira a fornecer uma visão global comum, além de dar suporte a um mecanismo de pesquisa similar em termos de desempenho e completude ao de um banco de dados distribuído tradicional.

3.1 Localização e balanceamento de dados

Um problema interessante em bancos de dados distribuídos e em sistemas P2P é como fragmentar e distribuir os dados (a carga) entre os nós que compõem a rede. Normalmente, as aplicações desenvolvidas sobre esse tipo de arquitetura tem pouco ou nenhum controle sobre o posicionamento dos dados (colocação). Assim, DHTs não preservam a localidade dos dados e acabam por descartar a proximidade das informações específicas às aplicações cliente [Keleher et al., 2002]. Uma abordagem usada por algumas redes de sobreposição estruturadas é a distribuição aleatória uniforme através de uma função de dispersão. Fragmentos de dados são atribuídos aleatoriamente a nós e um índice deve ser mantido para recuperá-los.

Algumas tabelas de dispersão distribuídas incluem técnicas simples de balanceamento de carga e, em alguns casos, como parte essencial de sua concepção através do uso, por exemplo, de dispersão consistente [Karger et al., 1997]. Todavia, a capacidade de balanceamento de carga varia muito e geralmente pode ser melhorada através da utilização de estratégias de balanceamento de carga que atendam a necessidades específicas. Isso pode incluir uma melhor divisão do espaço de identificadores entre os nós ou, mais genericamente, a aplicação de um mecanismo aprimorado de equilíbrio de carga arbitrária baseado na largura média dos objetos armazenados no sistema.

Outros tipos de DHT, em particular aquelas que dão suporte a consultas por intervalo, dependem inerentemente de mecanismos explícitos de balanceamento de carga, visto que a ordem de distribuição dos recursos armazenados é mantida e pode ser altamente distorcida.

Mesmo que estas abordagens alcancem um bom balanceamento de carga (um importante critério para sistemas distribuídos), elas carecem de desempenho para operações mais complexas de processamento de dados distribuídos que vão além de pesquisas que recuperam apenas um único fragmento de dados. Uma abordagem para resolver este problema em sistemas de banco distribuído é a fragmentação de dados.

Estratégias de balanceamento de carga baseados em fragmentação tentam equilibrar a distribuição real dos recursos entre os nós e não dependem de uma distribuição uniforme de recursos no espaço de identificadores, tornando-as particularmente adequadas para DHTs com suporte a consultas por intervalo que usam funções de dispersão que preservam a

ordem lógica dos itens armazenados.

3.2 Desafios e oportunidades

Geralmente, em qualquer aplicação para banco de dados, uma transação não costuma exigir que todas as colunas (atributos) das linhas (tuplas) de uma relação devam ser obtidas durante seu processamento. Quando uma relação é dividida em fragmentos de baixa granularidade que acumulam diversas colunas, as colunas armazenadas em um fragmento de dados que são irrelevantes a uma transação, isto é, que não são acessados pela transação, aumentam o custo de recuperação e processamento da transação, especialmente quando o número de linhas envolvidas na relação é muito grande.

Em um sistema de banco de dados centralizado e com hierarquia de armazenamento, isto acarreta muitos acessos ao armazenamento secundário. Em um sistema distribuído de gerenciamento de banco de dados, quando os atributos relevantes (ou seja, atributos acessados por uma transação) são fragmentos de dados variados e atribuídos a locais diferentes, há um custo adicional devido ao acesso remoto e transferência dos dados.

Sistemas de banco de dados freqüentemente acessam grandes quantidades de dados a fim de recuperar (obter) ou atualizar um número relativamente pequeno de valores determinados por uma aplicação, devido principalmente ao comprimento médio das linhas ser significativamente maior do que a quantidade de dados a serem recuperados ou modificados em uma linha.

Assim, características desejáveis de sistemas de gerenciamento de banco de dados distribuído a serem atingidas através da fragmentação são a acessibilidade e granularidade dos dados. Em outras palavras, cada local deve ser capaz de recuperar e processar operações com o mínimo de acesso a dados irrelevantes e/ou localizados em locais remotos. No caso de banco de dados distribuídos, o custo de processamento de transações é minimizado pelo aumento de operações locais de processamento, bem como pela redução da quantidade de acesso a dados irrelevantes e/ou dados que não são locais. Em geral, o objetivo das técnicas de fragmentação de dados é encontrar um esquema de fragmentação que satisfaça esta característica.

A fragmentação de dados melhora o desempenho de consultas e transações (por conseguinte, aplicações) através da redução da quantidade de dados irrelevantes que precisam

ser acessados e transferidos entre diferentes locais de um sistema distribuído. A decomposição de um objeto em fragmentos também possibilita processamento concorrente, visto que uma consulta pode acessar fragmentos de um mesmo objeto de forma simultânea.

É importante salientar que a fragmentação de dados deve ser implementada com cuidado, pois um número escasso de fragmentos pode levar a um desequilíbrio de carga e em demasiado acarretará no aumento do custo de pesquisa, já que vários fragmentos precisam ser recuperados a fim de responder a uma única consulta.

3.3 Fragmentação de dados

A fragmentação de dados permite a quebra de um objeto único em dois ou mais segmentos ou fragmentos. O objeto pode ser uma relação (tabela) ou um esquema. Cada fragmento pode ser armazenado em qualquer local de uma rede. Informações sobre a fragmentação dos dados podem ser armazenadas num catálogo distribuído de dados, o qual pode ser acessado para processar solicitações.

O objetivo da fragmentação de dados é dividir os dados em fragmentos menores de dados correlacionados que possam então ser armazenados em locais físicos diferentes. No modelo relacional, estratégias de fragmentação de dados, consistem em dividir uma tabela em fragmentos lógicos. São explorados três tipos de estratégias de fragmentação de dados, como ilustrado na figura 3.1: horizontal, vertical e híbrida (ou mista). Uma tabela fragmentada pode sempre ser recriada a partir de suas partes por combinação e associação.

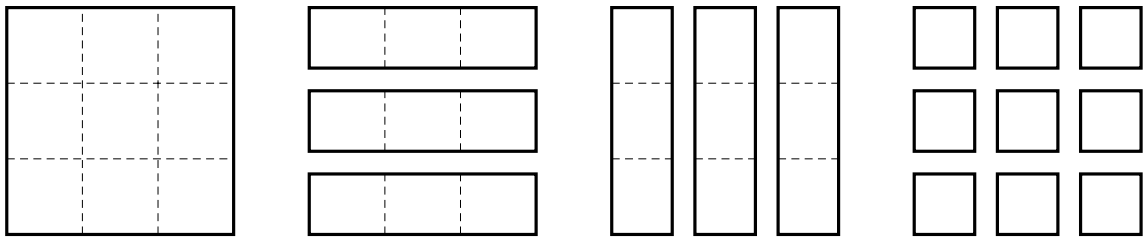

Figura 3.1: Estratégias de fragmentação.

A fragmentação horizontal refere-se à divisão de uma relação em subconjuntos (fragmentos) de tuplas (linhas). Cada fragmento pode ser armazenado em um nó diferente, e cada fragmento tem linhas distintas. Entretanto, todas as linhas têm os mesmos atributos

(colunas).

A fragmentação vertical refere-se à divisão de uma relação em subconjuntos de atributos (colunas). Cada subconjunto (fragmento) pode ser armazenado em um nó diferente, e cada fragmento tem colunas distintas, com exceção da coluna chave, que é comum a todos os fragmentos.

A fragmentação híbrida (ou mista) refere-se à combinação das estratégias horizontal e vertical, ou seja, uma tabela pode ser dividida em vários subgrupos horizontais (linhas), cada um com um subconjunto de atributos (colunas).

3.3.1 Fragmentação horizontal

A fragmentação horizontal [Ceri et al., 1982] divide uma tabela relacional em pedaços menores, chamados de fragmentos ou partições, de acordo com o intervalo de valores de um determinado atributo ou de uma combinação de atributos, chamada de chave de partição. De maneira formal, a fragmentação horizontal divide tuplas de uma relação em subconjuntos disjuntos e pode ser representada pela qualificação da seleção sobre a relação. A relação pode ser reconstruída unindo os subconjuntos.

A divisão horizontal reduz o número total de operações de leitura necessárias para acessar as linhas determinadas pelos valores da chave de partição fornecida em uma consulta.

Essa forma de fragmentação normalmente é útil se determinados intervalos de valores são muitas vezes tratados de forma conjunta. Por exemplo, os estudantes com códigos postais inferiores a 5000 são armazenadas na partição Sul, enquanto que os alunos com código maior ou igual a 5000 são armazenados na partição Norte. Uma visão com uma união ao longo de ambas as partições pode ser criada para fornecer uma visão completa de todos os alunos.

3.3.2 Fragmentação vertical

A fragmentação vertical [Navathe et al., 1984] divide os atributos de uma relação em grupos e pode ser representada por uma projeção qualificada sobre a relação. Para reconstruir a relação, é necessário que o atributo chave esteja incluído em cada grupo, e seus valores sejam duplicadas em cada fragmento da relação. A reconstrução da relação

pode ser feita juntando todos os fragmentos sobre o atributo de chave. A normalização [3] é um processo que envolve, inerentemente, fragmentação vertical.

Uma forma comum de fragmentação vertical é separar dados dinâmicos (lento de encontrar) de dados estáticos (fácil de encontrar) em uma tabela onde os dados dinâmicos não são usados tão freqüentemente quanto os dados estáticos. A criação de uma visão com a junção das duas tabelas recém-criadas restaura a tabela original com uma penalidade de desempenho, porém o desempenho aumenta para o acesso aos dados estáticos.

O objetivo da fragmentação vertical é criar fragmentos verticais de uma relação, de modo a minimizar o custo de acesso ao conteúdo dos dados durante o processamento de transações. Se os fragmentos correspondem aproximamente às necessidades de um conjunto de transações, então o custo de processamento de transações pode ser minimizado. Quanto mais fragmentos obtidos estiverem próximos das necessidades da transação, mais eficiente é o sistema. O caso ideal ocorre quando cada transação corresponde exatamente a um fragmento, ou seja, precisa apenas deste fragmento.

3.3.3 Fragmentação híbrida

Fragmentação híbrida [Navathe et al., 1995] é definida como o processo de aplicação simultânea de fragmentação horizontal e vertical em uma relação. A fragmentação híbrida pode ser realizada de duas maneiras: aplicando primeiramente a fragmentação vertical seguida da divisão horizontal dos fragmentos verticais (chamado de fragmentação VH), ou aplicando primeiramente fragmentação horizontal seguida da divisão vertical dos fragmentos horizontais (chamado de fragmentação HV). A fragmentação híbrida também pode ser abordada diretamente em termos de células dos dados, geradas pela grade (ou grelha) formada pelos esquemas de fragmentação horizontal e vertical.

A necessidade de fragmentação híbrida surge comumentemente em bancos de dados distribuídos, quando aplicações acessam subconjuntos de dados que são fragmentos verticais e horizontais das relações, estabelecendo a inevitabilidade de se processar de forma otimizada consultas ou transações que acessam esses fragmentos.

[3] A normalização é o processo utilizado para garantir que um modelo de dados esteja em conformidade com normas estabelecidas para evitar a duplicação de dados e para minimizar anomalias de atualização dos dados e envolve a estruturação de dados em um formato tabular.

3.4 Mapeamento entre modelos de dados

As camadas adjacentes inferiores do sistema proposto neste trabalho são compostas por dois diferentes níveis de abstração. A camada distribuída de armazenamento é baseada num modelo chave-valor, enquanto que a camada do banco de dados relacional segue, logicamente, um modelo relacional de dados. Faz-se necessário, portanto, um mecanismo de mapeamento que promova a integração de dados entre os dois modelos.

Um mapeamento é definido como o processo de traduzir uma operação definida em algum nível de abstração, dentro do SGBD, para as operações correspondentes definidas em outro nível de abstração. Nesta abordagem, é necessário traduzir as operações definidas no modelo relacional para operações no modelo chave-valor (e vice-versa).

Esse mapeamento é importante porque a camada subjacente (repositório de dados chave-valor) oferece apenas serviços, como busca e inserção, baseados em palavras-chaves e não provê funcionalidades semelhantes às presentes em um banco de dados relacional. Esse mapeamento deve permitir o armazenamento e representação, em termos do modelo chave-valor, de dados no modelo relacional.

A representação utiliza meta-informações para descrever e indexar dados, determinando critérios úteis para a formulação de consultas e demais operações definidas no modelo relacional. Esses meta-dados (ou meta-informações) são armazenados usando a camada de armazenamento do repositório de dados distribuído. A ausência de um componente central de controle no sistema torna o mapeamento particularmente desafiante já que se espera que o processo acompanhe a abordagem P2P do sistema sem introduzir pontos de centralização.

3.4.1 Modelo de mapeamento

Técnicas de mapeamento de modelos mudam a representação e/ou o nível de abstração usados em um modelo para outra representação e/ou o nível de abstração em outro modelo e são usadas para alcançar integração de dados. Em termos gerais, o mapeamento de modelos é usado para permitir que diferentes tipos de modelos sejam relacionados entre si.

Um modelo de mapeamento, como ilustrado na figura 3.2, é uma tradução de um tipo de modelo para outro tipo de modelo, que expressa a correspondência entre as ações

no domínio abstrato para outras ações equivalentes no domínio concreto. O modelo de mapeamento também expressa as restrições sobre as entidades modeladas.

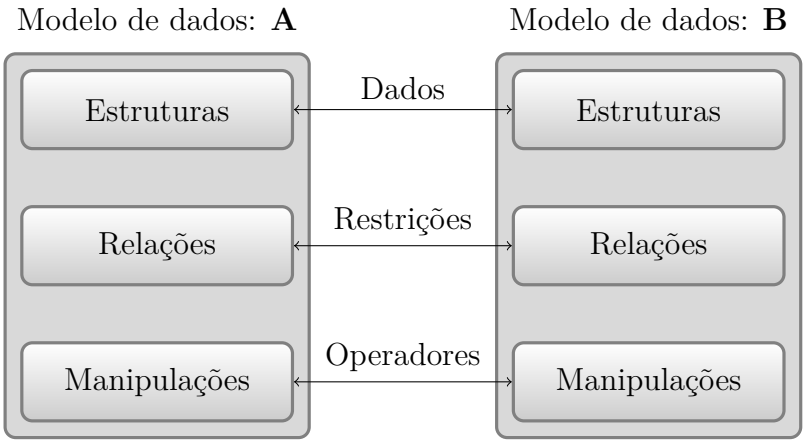

Figura 3.2: Mapeamento de modelos.

Na transição entre modelos de dados é necessário preservar as informações, operadores e relações. Isso requer que os modelos sejam equivalentemente representados pelo modelo de mapeamento. Dois modelos de dados são equivalentes se cada esquema em um modelo pode ser colocado em correspondência um para um com o esquema equivalente em outro modelo, concomitante ao fornecimento de conformidade entre os operadores de manipulação de dados definidos em cada modelo.

Um modelo de mapeamento pode ser decomposto em termos de mapeamento e transformação. Mapeamento denota o mecanismo abstrato de conversão de um modelo para outro e transformação, a aplicação do mapeamento em software. Este trabalho enfoca os aspectos de transformação do modelo de mapeamento, essencialmente na transformação dos dados entre o modelo relacional e chave-valor.

3.5 Trabalhos relacionados

Os primeiros trabalhos a considerarem funcionalidades complexas de banco de dados sobre redes P2P de sobreposição foram [Gribble et al., 2001] e [Bernstein et al., 2002]. Ambos os trabalhos propõem visões iniciais sobre a combinação dos campos de pesquisa P2P e banco de dados. Enquanto o primeiro se concentra nas oportunidades e desafios, e discute o problema da localização dos dados, o último trata dos problemas da integração de dados.

Do ponto de vista de banco de dados, os primeiros passos para a distribuição foram tomados por bancos de dados particionados, onde os dados (e, opcionalmente, o controle)

são distribuídos através de uma rede de computadores. Normalmente, os recursos de particionamento são funcionalidades ou extensões incorporadas a bancos de dados tradicionais. Semelhante às técnicas de agrupamento (clusterização) destes banco de dados tradicionais, existem algumas propostas de protocolos P2P que fornecem mecanismos para indicar que um grupo de dados deve ser mantido num mesmo servidor de armazenamento, a fim de preservar a proximidade entre dados similares. Algumas permitem um controle sobre a colocação dos dados, tais como SkipNet [Harvey et al., 2003a], enquanto outras colocam os dados em ordem lexicográfica, baseada nas chaves de armazenamento, como o Chord# [Schütt et al., 2006]. Quando a estratégia de ordem lexicográfica é usada, é possível gerar chaves de armazenamento com as propriedades de agrupamento desejadas.

Um trabalho estreitamente relacionado que também é motivado por funcionalidades de banco de dados para sistemas P2P é o PIER [Huebsch et al., 2003, Huebsch et al., 2005]. Além de dar suporte a consultas sobre dados homogêneos, o PIER também sugere o uso de uma DHT na camada física. Mas, ao contrário do aqui proposto, a DHT não é decomponível do sistema. Na camada de armazenamento, os fragmentos das relações são distribuídos entre os nós participantes seguindo a idéia de fragmentação horizontal dos dados. Tuplas são anotados (etiquetadas) pelo nome da relação, os nomes de atributos e tipos dos atributos. Um fragmento (ou partição) de atributo pode ser selecionado para a construção de um índice primário. Além disso, também são suportados índices secundários que contêm somente referências. Para consultas por intervalo, o PIER utiliza a técnica proposta em [Ramabhadran et al., 2004]. Por conseguinte, o PIER suporta consultas SQL padrão.

Um estudo de visão mais abrangente das tecnologias de distribuição de dados em redes P2P, incluindo abordagens baseadas em DHT, pode ser encontrado em [Androutsellis-Theotokis and Spinellis, 2004]. Mas, o estudo avalia diversas abordagens a partir de um ponto de vista de redes e perde de perspectiva as questões relacionadas a banco de dados. Devido à ausência das abordagens de modelagem de dados advindas de banco de dados, a análise do processamento de consultas é baseado em consultas demasiadamente simples. Dependendo do modelo de dados, a estreita relação entre a distribuição de dados e abordagens de processamento de consultas exige um olhar mais atento. No entanto, o estudo dá informações valiosas sobre os sistemas existentes e também discute questões relacionadas à segurança. Uma discussão sobre questões relativas ao gerenciamento de dados XML

em um contexto P2P, incluindo indexação, clusterização, replicação e processamento de consultas pode ser encontrada em [Koloniari and Pitoura, 2005].

CAPÍTULO 4

CAMADA DE INTEGRAÇÃO E DISTRIBUIÇÃO

Este capítulo apresenta a camada de integração e distribuição da abordagem proposta. Primeiramente, são abordadas, de maneira resumida, as vantagens da utilização de uma DHT nesta camada, a fim de mostrar como alavancar a escalabilidade, transparência de localização, buscas, e outras garantias, como robustez e disponibilidade providas pela DHT, além de brevemente listar os pressupostos e requisitos da DHT. As demais seções tratam das estratégias de fragmentação e mapeamento do sistema.

O desafio mais importante é como modelar os dados de maneira genérica e flexível, pois este fator também concerne o modelo e distribuição de dados. Do ponto de vista conceitual, ambos aspectos são combinados através de um modelo de dados comum baseado no OEM [Papakonstantinou et al., 1995]. O OEM é um modelo de dados concebido para o intercâmbio de dados semi-estruturados entre bancos de dados, podendo ainda ser utilizado de forma genérica para modelar informações como metadados. Este modelo é adotado devido a sua capacidade em modelar dados estruturados, semi-estruturados e heterogêneos. Esta combinação permite criar um modelo de representação e gestão de dados que cumpra os requisitos de universalidade, flexibilidade e extensibilidade.

4.1 Repositório de dados distribuídos (DHT)

Conforme descrito na seção 1.3, a arquitetura proposta é divida em camadas. Na camada inferior, a DHT é utilizada como substrato de armazenamento. Todas as operações realizadas acima da camada da DHT apenas usam funcionalidades da mesma. A DHT é responsável por gerenciar seus integrantes (nós), ou seja, lidar com situações dinâmicas como a entrada, saída e falha de nós, além dos mecanismos de consultas e roteamento de mensagens. Quanto aos desafios apresentados anteriormente, as DHT se revelam vantajosas, haja vista que fornecem um alicerce para alcançar escalabilidade em termos de participantes, montantes de dados e processamento de consultas.

Uma vez que é utilizada somente a funcionalidade padrão da DHT, a arquitetura não se limita a um sistema específico. Apesar disso, algumas partes utilizam funcionalidades

especiais que não são apresentadas, por padrão, por todos os sistemas. Entretanto, estas são opções para alavancar a eficiência. O esquema de integração e distribuição proposto neste trabalho pode ser implementado em cima de qualquer sistema padrão de DHT. As funcionalidades e características que são desejáveis, mas não obrigatórias, é o apoio a consultas por abrangência e recursos de balanceamento de carga. Note-se que assumir a existência desses recursos não é irreal, haja vista que vários sistemas de DHT provêem estes e outros recursos sofisticados. A motivação é integrar paradigmas de processamento inovadores a eficientes técnicas de processamento já disponíveis na DHT. Neste meio tempo, quase todas DHT populares suportam consultas por abrangência, quer pela sua natureza [Bharambe et al., 2004, Gao and Steenkiste, 2004, Datta et al., 2005, Schütt et al., 2007] ou extensão [Sahin et al., 2004, Liau et al., 2004, Zheng et al., 2006, Ramabhadran et al., 2004].

O Scalaris [Berlin and onScale solutions GmbH, 2010] é uma DHT que suporta nativamente consultas abrangência, devido a organização das chaves em ordem lexicográfica. Apesar deste trabalho propor uma arquitetura que possa fazer uso de qualquer DHT disponível, o Scalaris é utilizado como base para o desenvolvimento do sistema . Há várias razões para isso, que serão abordadas ao longo desse trabalho, mas existe uma especificamente, que é o fato de o Scalaris suportar todos os requisitos presentes neste trabalho, tais como consultas por abrangência, técnicas sofisticadas de balanceamento, transações e consistência forte.

4.2 Estratégia de fragmentação

Para proporcionar um melhor balanceamento de carga e instigar a utilização de recursos e replicação, os dados das relações armazenadas no sistema proposto podem ser fragmentados dinamicamente a partir de uma abordagem horizontal, vertical ou híbrida de fragmentação.

O esquema de fragmentação é construído de maneira a formar uma grade composta de células. As células da grade são obtidas em simultâneo com a geração de fragmentos horizontais e verticais. Cada célula corresponde exatamente a um fragmento vertical e horizontal. A granularidade dos dados passa a ser cada valor dos campos de uma tabela e pode ser controlada pelo usuário mediante o número desejado de fragmentos.

Dada uma relação, os subconjuntos de suas tuplas dão forma a fragmentos horizontais definidos por fragmentação horizontal; os subconjuntos de seus atributos dão forma a fragmentos verticais definidos por fragmentação vertical. A aplicação simultânea de fragmentação vertical e horizontal na relação dá origem a uma grade constituída por um conjunto de células. As células da grade compõem uma compartimentação imparcial da relação.

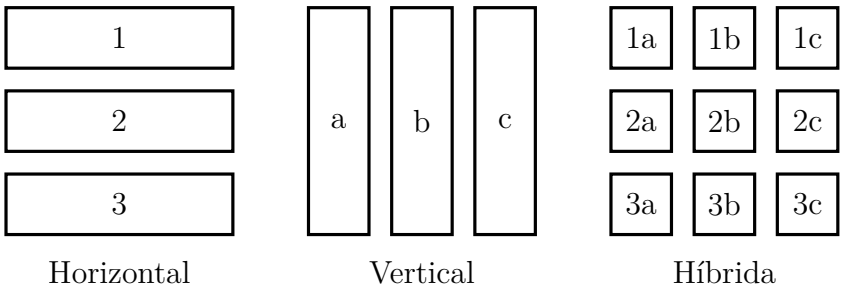

Figura 4.1: Representação das células.

A grade é criada mediante a aplicação de ambos os esquemas de fragmentação horizontal e vertical na relação. Como ilustrado na figura 4.1, seja H = (1, 2, ..., n) o conjunto de fragmentos horizontais e V = (a, b, ..., m) o conjunto de fragmentos verticais de uma relação. A grade gera um conjunto de células no qual cada célula pertence exatamente a um fragmento horizontal e a um fragmento vertical da relação. O conjunto de células da grade é representado como (1a, 1b, ..., 1m; 2a, 2b, ...,2m; ...; na, nb, ..., nm).

4.3 Estratégia de mapeamento

Tendo por base a perspectiva de concepção do sistema, a abordagem proposta é baseada no conceito de independência de dados com a fragmentação, replicação e transparência de localização. Dito de outro modo, o sistema fornece uma visão relacional e uniforme dos dados, independentemente da estrutura e modelo de dados do armazenamento subjacente. A fim de apoiar a interoperabilidade (intercâmbio e uso das informações) entre as camadas banco de dados (lógico) e de armazenamento (físico), é necessário definir um processo de integração e mapeamento de dados que torne completamente transparente o armazenamento e a heterogeneidade dos modelos de dados.

O processo é essencialmente centrado na transferência e conversão de dados entre as camadas e seus respectivos modelos de dados. No nível lógico, os dados são representados

e manipulados em termos do modelo relacional, enquanto no nível físico os dados são acessados e armazenados em termos de um modelo semi-estruturado. A informação descrita em um dos modelos de dados é mapeada em uma descrição equivalente da mesma informação no modelo de dados oposto. Da mesma forma, as consultas sobre as informações em um modelo são convertidas em pedidos sobre as mesmas informações no outro modelo. Até certo ponto, o mapeamento descreve como armazenar e recuperar dados relacionais como dados semi-estruturados.

Além do intercâmbio de informações, mas ainda no âmbito da interoperabilidade, se estende um conjunto de dimensões associadas: manejabilidade, desempenho e ubiquidade. O processo de intercâmbio de dados deve servir de base para um processamento de dados eficiente e escalável. Por exemplo, uma combinação de técnicas de agrupamento e de fragmentação pode gerar melhorias significativas na escalabilidade e desempenho do sistema, reduzindo custos de comunicação e armazenamento. As várias instâncias do banco de dados devem ser capazes de serem integrados de maneira coerente e de localizar de forma transparente todos os objetos de dados armazenados no sistema, sem incorrer em maior complexidade e mantendo a autonomia. Por exemplo, uma instância de banco de dados deve ser capaz de manipular partes de um objeto lógico maior, sem interferir com outras instâncias usando diferentes partes do mesmo objeto lógico.

Para facilitar e viabilizar os objetivos de integração de dados, um modelo comum de dados é adotado como estrutura para a integração. O modelo comum atua como uma camada de mapeamento que une os diferentes modelos de dados dos componentes do sistema, fornecendo um substrato flexível para o processamento de dados. Acima de cada componente existe um invólucro que, sob demanda, converte de um lado para o outro os objetos de dados subjacentes ao modelo comum. Os dados não precisam ser armazenados fisicamente no formato comum. O modelo comum é utilizado para integração, processamento e consultas. Um modelo de dados mais primitivo tem a vantagem de simplificar as operações de transformação e dispersão dos dados e ao mesmo tempo manter um elevado nível de decomposição e transparência. Esta representação intermediária baseia-se numa variação do modelo de dados OEM (*Object Exchange Model*).

4.4 Modelo Phoenix de intercâmbio de objetos

A representação comum é uma extensão do OEM [Papakonstantinou et al., 1995], denotada como POEM (*Phoenix Object Exchange Model*), no qual os objetos são identificados por valores. O OEM é um modelo de dados leve, auto-descritivo e de tipagem dinâmica que fornece um substrato capaz de representar praticamente qualquer estrutura de dados. Ele permite uma modelagem simples e flexível de estruturas de dados complexas, utilizando conceitos como identidade de objeto e aninhamento. No OEM cada objeto tem um identificador único (*id*) e as relações entre os objetos são representadas usando identificadores como sub-objetos ou valores de atributos. No modelo relacional, por outro lado, as relações entre os dados são representadas usando o conceito de chaves e chaves estrangeiras. Ou seja, enquanto os relacionamentos OEM são baseados em identificadores, os do modelo relacional são baseados em valores. Para resolver esta disparidade, o OEM é estendido com a noção de *chave*, ou seja, um conjunto de sub-objetos que identificam um objeto.

Um objeto POEM é uma quíntupla $\langle$*id*, *rótulo*, *tipo*, *valor*, *chave*$\rangle$. O *id* é um identificador exclusivo para o objeto, e o *rótulo* descreve o que o objeto representa. O *tipo* de dado de um objeto pode ser primitivo (também conhecido por nativo ou básico) ou composto. O *valor* de um objeto primitivo é uma instância de um dos tipos de dados escalares, enquanto que o *valor* de um objeto composto é um conjunto de referências a objetos $\{id_1, id_2, \ldots, id_n\}$. A *chave* de um objeto composto é um par $\langle contextoId, subObjs\rangle$, onde *contextoId* é um identificador de objeto que define o contexto em que a chave é definida, e *subObjs* é um subconjunto do *valor* do objeto que pode identificar unicamente o objeto dentro do contexto.

O exemplo de uma composição de objetos POEM é apresentado na Figura 4.2. Os objetos representam um conjunto de livros. O objeto identificado por $livro_R$ representa a entidade *livros* e é composto por conjunto de referências a objetos *livro*. Cada objeto descrito como *livro* é composto por referências à objetos que descrevem o título, autor e identificador de cada livro.

Para extrair o *valor* e o *rótulo* de um objeto, algumas funções são introduzidas. A função `valorObj`(id) retorna o *valor* associado com o objeto identificado por *id*, e $rotuloObj(id)$ retorna o *rótulo* associado ao objeto identificado por *id*. Por exemplo, con-

$\langle livro_R$, livros, conjunto, $\{livro_1, livro_2, \ldots, livro_n\}\rangle$
  $\langle livro_1$, livro, conjunto, $\{id_1, título_1, autor_1\}\rangle$
    $\langle id_1$, id, inteiro, 1$\rangle$
    $\langle título_1$, titulo, string, ‘‘Iracema’’$\rangle$
    $\langle autor_1$, autor, string, ‘‘José de Alencar’’$\rangle$
  $\langle livro_2$, livro, conjunto, $\{id_2, título_2, autor_2\}\rangle$
    ...
  $\langle livro_n$, livro, conjunto, $\{id_n, título_n, autor_n\}\rangle$
    $\langle id_n$, id, inteiro, ...$\rangle$
    ...

Figura 4.2: Estrutura de objetos *livros*

siderando os objetos POEM representados na Figura 4.2, `valorObj`$(título_1) =$ “$Iracema''$, `valorObj`$(livro_1) = \{id_1, título_1, autor_1\}$, e `rótuloObj`$(título_1) = título$.

A estrutura aninhada de objetos POEM também pode ser representada por um grafo dirigido, rotulado e valorado, como mostrado na Figura 4.3. Nessa representação, cada vértice (ou nó) é rotulado com seu respectivo *id*, que identifica unicamente o vértice em um grafo, e as arestas são marcadas com o *rótulo* de seus respectivos objetos. As arestas rotuladas conduzem informação semântica sobre as relações entre os objetos. Assim, dependendo do número de arestas convergentes, um único objeto pode servir a vários papéis. Objetos compostos do tipo de dado *conjunto* são vértices internos com grau de saída (número de arestas divergentes) superior a zero, enquanto os objetos primitivos são vértice folhas (sumidouros com grau de saída nulo) que estão associados a valores de dados.

Uma aplicação útil da representação gráfica é o conceito de expressões de caminho. Um caminho, entre outras coisas, pode ser usado para navegar pela hierarquia de objetos. Em um grafo POEM, uma expressão de caminho é uma seqüência de rótulos das arestas, como em */livros/livro/id*. Aqui, a barra comum (“/”) mais à esquerda representa o nó raiz, enquanto as demais denotam o percursso de uma aresta. Uma expressão de caminho permite, de maneira prática, a obtenção de um conjunto de objetos acessíveis seguindo uma seqüência de rótulos. Para representar essa idéia, a seguinte notação é introduzida: $o[\![c]\!]$ é o conjunto de objetos POEM obtidos ao percorrer o caminho c a partir do nó identificado por o. Se o não é especificado, a travessia começa a patir da raiz do grafo. Por exemplo, $livro_R[\![livro/título]\!] = \{\langle título_1$, título, string, “Iracema”$\rangle$, ..., $\langle título_n$, título, string, “Título$_n$”$\rangle\}$. Dados dois identificadores de objeto o_1 e o_2,

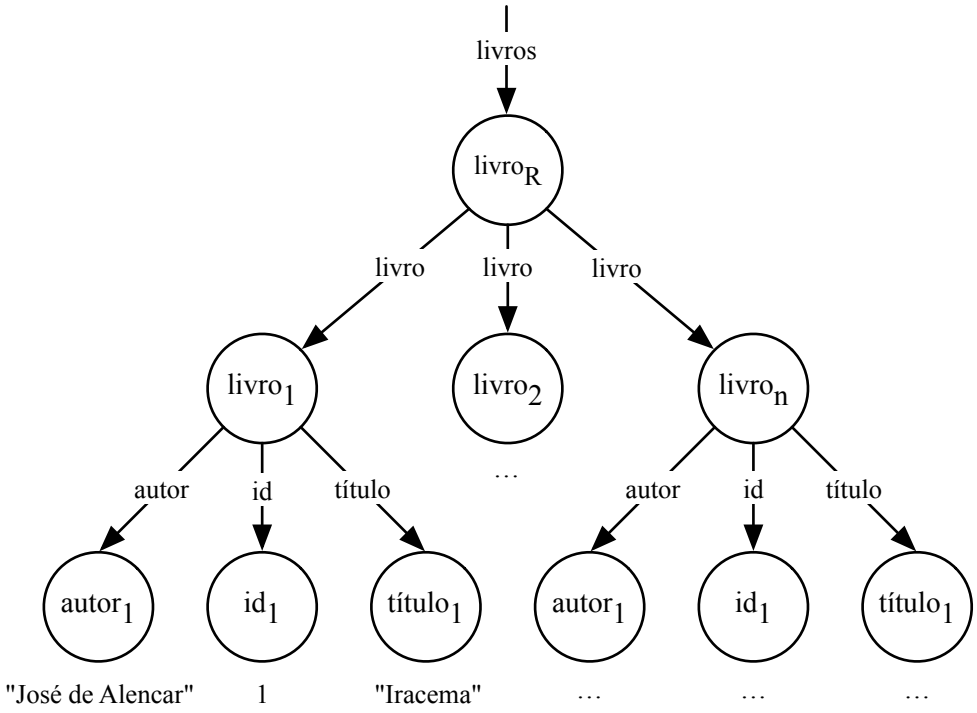

Figura 4.3: Grafo de dados POEM

denotamos por $caminho(o_1, o_2)$ o caminho definido pelos rótulos das arestas percorridas para chegar a o_2 a partir de o_1. Por exemplo, $caminho(livro_R, título_1) = livro/título$.

Apresentada a noção de grafos POEM, é possível dar uma definição formal para chaves POEM. Seja $\langle o_1,$ *rótulo*, *tipo*, *valor*, *chave*$\rangle$ um objeto, uma chave é dada por um par $(o_2, \{sub_1, \ldots, sub_n\})$. A chave especifica que o valor do conjunto de pares $\{(\texttt{rótuloObj}(sub_1), \texttt{valorObj}(sub_1)), \ldots, (\texttt{rótuloObj}(sub_n), \texttt{valorObj}(sub_n))\}$ identifica unicamente o_1 entre os objetos em $o_2[\![caminho(o_2, o_1)]\!]$. Informalmente, os valores dos sub-objetos identificam o_1 unicamente entre aqueles alcançados pela travessia do mesmo caminho de o_1 no subgrafo com raiz em o_2. Por exemplo, a chave no objeto $\langle livro_1$, livro, conjunto, $\{id_1, título_1, autor_1\}$, $(livro_R, \{id_1\})\rangle$ determina que $\{\langle \text{id}, 1\rangle\}$ identifica unicamente o objeto $livro_1$ entre o conjunto de objetos `livro` do subgrafo com raiz em $livro_R$. Ou seja, não existem dois objetos `livro` com um mesmo atributo `id`.

A adoção do POEM como interface entre os modelos relacional e chave-valor exige que tanto o banco de dados, como o serviço de armazenamento de dados, estejam associados a mapeamentos para converter seus dados subjacentes a objetos POEM e de volta ao seu formato original.

4.4.1 Mapeamento relacional

A transformação de dados do modelo relacional para objetos POEM é um processo simples, baseado no mapeamento dos conceitos básicos do modelo relacional para uma hierarquia de objetos POEM. Esses conceitos básicos do modelo relacional são a relação, domínio, tupla, atributo e chave. Uma tabela é uma representação concreta de uma relação, o domínio é equivalente ao tipo de dados (por exemplo, inteiro, string, etc), uma tupla (também chamado de registro ou linha) é um conjunto desordenado de valores de atributo e atributo (também chamado campo) é um par ordenado de nome do atributo e nome do tipo. Assim, um banco de dados relacional consiste em um conjunto de tabelas, onde cada tabela é um conjunto de registros. Um registro, por sua vez, é formado por um conjunto fixo de campos (também chamados de colunas) e cada campo é um par nome-valor de um determinado tipo.

Relação *livros*		
Id	***Título***	***Autor***
1	Iracema	José de Alencar
2	O Seminarista	Bernardo Guimarães
3	Inocência	Visconde de Taunay
...	...	...
Id_n	$Título_n$	$Autor_n$

Tabela 4.1: Relação *livros*, chave primária *Id*.

Os conceitos relação, tupla e atributo são relacionados por uma hierarquia de composição. Esta hierarquia pode ser diretamente representada por objetos POEM num contexto onde um objeto raiz descreve uma relação e as chaves são definidas para objetos que simbolizam tuplas. Considerando a relação da Tabela 4.1, um exemplo deste mapeamento é a hirarquia de objetos apresentada na Figura 4.3. Nela, o objeto identificado por $livro_R$ representa a relação *livros* e é constituído de um rótulo `livros`, do tipo `conjunto` e tem um conjunto de *id* como seu valor, cada qual associado a uma tupla da relação. É importante notar que cada objeto na hierarquia POEM possui um *id* único e todo relacionamento entre os objetos são representados por referências ao *id*.

4.4.2 Mapeamento chave-valor

Um mapeamento de POEM para pares chave-valor é definido para satisfazer uma única condição: cada par deve ter uma *chave* única e distinta para evitar colisões no espaço de chaves e outras formas de conflito. Para distinguir os componentes dos pares chave-valor, os termos *chave de armazenamento* e *valor de armazenamento* são usados para indicar os elementos respectivos de um par. Dado que objetos POEM têm identificadores distintos e carregam informação de identificação baseado no valor, há uma série de mapeamentos possíveis de POEM em pares chave-valor.

A estratégia mais simples é mapear cada objeto POEM a um par chave-valor, em que a chave de armazenamento consiste do *id* e os campos restantes do objeto compõem o valor de armazenamento. Dado que a chave de armazenamento determina a distribuição dos dados entre os servidores de armazenamento, esta estratégia de mapeamento distribui aleatoriamente os objetos, uma vez que os identificadores consistem de valores arbitrários atribuídos aos objetos. Distribuição randômica de dados pode ser desejável em algumas aplicações, mas geralmente esse não é o caso para aplicações de banco de dados. Clusterização (agrupamento) de índices é reconhecido como uma forma eficaz de aumentar o desempenho de SGBDR. O conceito de agrupar dados que normalmente são acessados em conjunto também se aplica em um ambiente distribuído, mas com interesse em mantê-los num mesmo servidor de armazenamento.

Uma forma de facilitar a tarefa de agrupamento de pares chave-valor é definir uma estratégia de mapeamento na qual a chave de armazenamento reflete a estrutura dos dados, mas sem atenuar a condição de identificação única. Uma possível abordagem é construir a chave de armazenamento combinando uma expressão de caminho a alguns elementos chave do objeto. Por exemplo, um mapeamento para gerar um par para cada valor do atributo `autor`, pode ser definido da seguinte forma: obter todos os objetos que podem ser alcançados a partir do caminho */livros/livro/autor*, e para cada um, gerar um par em que a chave de armazenamento é composta por "/livros/livro/autor/" concatenado ao valor objeto *livro* correspondente. Denotando por "$\circ$" um operador de concatenação, esse mapeamento de pares chave-valor pode ser expresso da seguinte forma:

$$
\begin{aligned}
m_1 = \{ & /livros/livro/autor/ \circ \texttt{valorObj}(\$id),\ \texttt{valorObj}(\$a)) \mid \\
& \langle\ \$b,\ \texttt{livro},\ \_,\ \_,\ (livro_R,\ \{\$id\})\ \rangle \text{ em } [\![/livros/livro]\!], \\
& \langle\ \$a,\ \texttt{autor},\ \_,\ \_,\ \_)\ \rangle \text{ em } \texttt{valorObj}(\$b)\ \}
\end{aligned}
$$

Nesta expressão um identificador precedido por $ denota uma variável e "_" denota espaços reservados a dados que não são significativos para a definição do mapeamento. Esse mapeamento especifica uma iteração sobre objetos `livro` em $[\![/livros/livro]\!]$. De cada objeto *obj* neste conjunto, o *id* é extraído e atribuído à variável $\$b$ e o seu sub-objeto chave é atribuído à variável $\$id$. As referências que compõem o objeto *obj* então são considerados para obter aquele com rótulo `autor`. O *id* do objeto é então atribuído à variável $\$a$. A chave de armazenamento é construída pela concatenação do caminho $/livros/livro/autor/$ com o valor do objeto $\$id$ (referente ao atributo identificador), e o valor de armazenamento consiste no valor do objeto `autor` identificado por $\$a$. O resultado do mapeamento m_1 sobre os objetos da Figura 4.2 é o conjunto de pares {($/livros/livro/autor/1$, "José de Alencar"), ($/livros/livro/autor/2$, "Bernardo Guimarães"), ..., ($/livros/livro/autor/n$, "$autor_n$")}. Mapeamentos semelhantes podem ser definidos para a geração de pares para os atributos `título` e `id`. Esta estratégia de mapeamento resulta em uma fragmentação completa da relação, em que cada célula da tabela é armazenada individualmente em um par chave-valor.

A generalidade do modelo POEM também permite a definição de outros mapeamentos de maior granularidade. Por exemplo, a fragmentação horizontal da relação *livros*, em que cada tupla corresponde a um par chave-valor pode ser obtido da seguinte forma.

$$
\begin{aligned}
m_2 = \{ & /livros/livro/ \circ \texttt{valorObj}(\$id), \\
& \{(\texttt{rótuloObj}(\$field),\ \texttt{valorObj}(\$field)) \mid \$field \text{ em } \texttt{valorObj}(\$b)\} \\
& \mid\ \langle\ \$b, \texttt{livro}, \_, \_, (livros_R, \{\$id\})\rangle \text{ em } [\![/livros/livro]\!]\}
\end{aligned}
$$

Neste mapeamento, a variável $\$b$ itera sobre todos os objetos `livro`. Para cada um deles, o seu objeto `id` é associado com a variável $\$id$, e o valor de $\$id$ é usado para compor a chave de armazenamento. O valor de armazenamento é obtida iterando sobre cada objeto que está no valor de um objeto `livro`. De cada um deles o rótulo e o valor são extraídos para compor um conjunto de pares $(rótulo, valor)$ no valor de armazenamento. O resultado deste mapeamento sobre os objetos POEM na Figura 4.2 é o conjunto {($livros/livro/1$, {(id, 1), (título, "Iracema"), (autor, "José de Alencar")}), ($livros/livro/2$, {(id, 2), (título, "O Seminarista"), (autor, "Bernardo Guimarães")}), ...}.

A fragmentação vertical (ou em coluna) de uma relação também pode ser definida da mesma forma. As diferentes formas de mapeamentos de POEM para pares chave-valor

podem ser definidas para cada função, ou para diferentes níveis de uma hierarquia de composição. A fim de permitir que diferentes SGBDR possam acessar os dados compartilhados, a construção da chave e valores de armazenamento também deve ser compartilhada entre eles.

A possibilidade de definir os níveis de granularidade da fragmentação resulta em uma redução no número de objetos que devem ser trocados (possivelmente através da rede) entre as camadas do sistema. A escolha da fragmentação que é mais adequada para uma determinada aplicação depende da sua carga de trabalho. Os experimentos apresentados no capítulo 6 mostram que a fragmentação pode ter um grande impacto sobre o desempenho do sistema.

CAPÍTULO 5

PHOENIX: BANCO DE DADOS RELACIONAL NA NUVEM

Este capítulo apresenta e detalha uma visão geral da arquitetura global do sistema Phoenix, bem como os aspectos relevantes que foram considerados no processo de desenvolvimento do protótipo. Há duas questões principais a serem consideradas para apoiar esta arquitetura. A primeira é a interface entre a nuvem e o serviço de armazenamento de dados, que envolve não só a transformação e mapeamento dos dados, mas também a distribuição dos dados entre os dispositivos de armazenamento. A segunda é a funcionalidade esperada que o serviço de armazenamento de dados deve fornecer para o sistema de banco de dados a fim de apoiar a arquitetura proposta.

5.1 Visão geral

Na arquitetura proposta, ilustrada na Figura 5.1, o serviço de armazenamento de dados na nuvem é implementado por um repositório de dados distribuído e pode ser visto como um espaço de armazenamento compartilhado entre um conjunto de SGBDR. Enquanto o armazenamento de dados oferece escalabilidade, replicação de dados e consistência transacional forte, o SGBDR fornece uma linguagem de consulta de alto nível para acessar e tratar os dados armazenados na nuvem. Ou seja, o serviço de armazenamento é utilizado como um alicerce para a estruturação e organização de uma base de dados relacional e não somente para buscas baseadas em chave.

Nesta arquitetura, o esquema de mapeamento define como criar pares de chave-valor em termos dos dados a serem armazenados. Em virtude da abordagem de independência de dados do modelo relacional, a estratégia de armazenamento físico pode ser implementada sem alterar a representação lógica dos dados. Logo, o mapeamento é usado para prover regras que possibilitem o fluxo de dados entre o nível lógico e físico do sistema, seguindo a estratégia delineada na seção 4.4.

Em conjunto com a estratificação dos níveis de abstração do banco de dados relacional, esse enfoque permite uma convergência à execução do repositório de dados distribuído como estrutura de armazenamento físico do sistema. A fim de recuperar os dados das

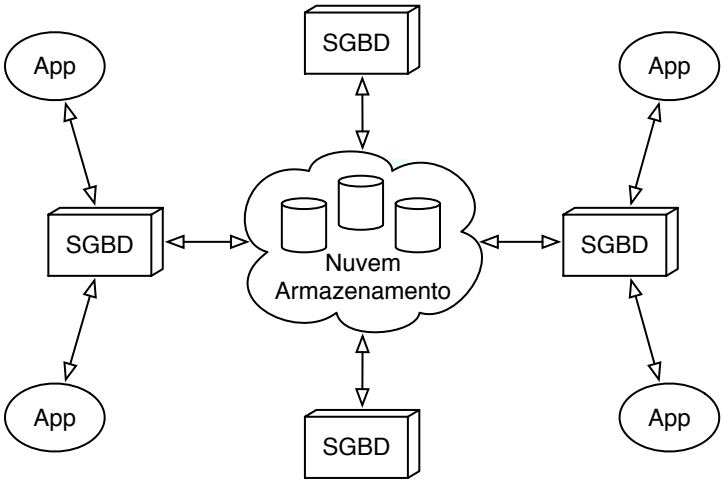

Figura 5.1: Visão geral do sistema

relações, as transformações aplicadas dentro da estratégia de armazenamento físico devem preservar informações essenciais à representação dos dados no modelo relacional, tais como nomes das relações, atributos e chaves.

Concomitantemente, mantêm-se componentes independentes através do uso fiel da interface de programação (API) definida pelo repositório, tornando a solução decomponível do repositório subjacente. Essa abordagem em camadas simplifica a manutenção e facilita a adição de melhorias ao sistema, graças a alta coesão das camadas, bem como a capacidade para alternar diferentes implementações das interfaces de camada.

5.2 Princípios da arquitetura

Arquiteturas baseadas em princípios de P2P têm sido amplamente aplicadas para melhorar os mecanismos subjacentes das plataformas de armazenamento na nuvem. DHTs e sistemas baseados em DHTs solucionam problemas que são comuns a vários sistemas distribuídos, tais como replicação e localização dos dados, sem a necessidade de esforço adicional por parte das aplicações que as usam. Esses sistemas também fornecem uma interface genérica, que facilita sua adoção como um substrato de armazenamento para uma ampla variedade de aplicações. A interface consiste em três operações:*put(chave,valor)*, que armazena o *valor* associado a uma *chave* no nó da rede P2P responsável pelo índice gerado pela função de espalhamento a partir da *chave*; e as operações *get(chave)* e *delete(chave)*, que recuperam e removem o *valor* associado a uma determinado *chave*.

Estes sistemas fornecem uma infra-estrutura de armazenamento quase universal de-

vido à sua aplicabilidade a uma ampla variedade de cenários. Embora estes sistemas forneçam uma série de funcionalidades desejáveis tais como descentralização, redundância, adaptabilidade, auto-organização e baixo custo operacional, existem dois recursos adicionais que são imprescindíveis a fim de apoiar sua utilização como um componente de armazenamento distribuído de um SGBD tradicional: localização (posicionamento) dos dados, e suporte a transações com consistência forte.

5.2.1 Localização dos dados

Considere novamente o exemplo da Figura 4.3 e as regras de mapeamento e fragmentação baseada em células definidas pelo mapeamento m_1, em que a chave de armazenamento é construído pela concatenação do caminho */livros/livro/autor/* com o valor do atributo `id`. Se os dados são colocados em ordem lexicográfica, pares contendo valores `autor` tendem a ser armazenados nos mesmos ou em servidores de armazenamento próximos, uma vez que todos possuem expressões de caminho similares, apesar de distintos devido ao valor do atributo identificador (`id`). Por outro lado, é provável que os pares que representam os atributos que compõem uma tupla não sejam armazenados juntos uma vez que chaves de armazenamento como */livros/livro/autor/*1 e */livros/livro/título/*1 não estão estreitamente próximas em ordem lexicográfica. Assim, este mapeamento define uma distribuição dos dados orientada a colunas.

A fim de definir uma distribuição de dados orientada a tuplas e com fragmentação baseada em células, a chave de armazenamento deve ser construída pela concatenação do caminho */livros/livro/* com o valor do atributo `id`, seguido por */autor*. Neste caso, as chaves */livros/livro/*1*/autor* e */livros/livro/*1*/título* estão em ordem lexicográfica. Esta técnica demonstra que a abordagem de mapeamento POEM para pares chave-valor permite não só definir diferentes níveis de granularidade para a fragmentação de dados, mas também diferentes formas de agrupar fragmentos nos servidores de armazenamento, quando o armazenamento subjacente mantém os dados em ordem. A capacidade de controlar a localização do conteúdo em um armazenamento de dados distribuída oferece uma série de vantagens para recuperação de dados, incluindo maior disponibilidade, desempenho, manejabilidade e segurança [Harvey et al., 2003a].

5.2.2 Transações e consistência

Em sistemas de banco de dados tradicionais, o serviço gerenciador (ou coordenador) de transações trabalha com o mecanismo de armazenamento a fim assegurar as propriedades de atomicidade, consistência, isolamento e durabilidade (ACID). O mesmo nível de abstração, fornecido pelo gerenciador de transações para os processos que acessam simultaneamente dados armazenados em um único servidor, deve ser fornecido pela plataforma de armazenamento de dados em nuvem. Devido a estreita ligação entre a manutenção de consistência forte entre as réplicas em um armazenamento distribuído e acessos concorrentes a um conjunto de itens de dados, é natural que os dois problemas sejam resolvidos pela camada de armazenamento físico.

Ademais, ao se abstrair o conceito de transações, é possível que vários SGBD, rodando em servidores diferentes, possam acessar o mesmo armazenamento de dados, fornecendo assim um serviço de banco de dados relacional na nuvem. A capacidade de manter a localidade dos dados está intimamente relacionada com a capacidade de dar suporte a consistência forte em um sistema distribuído. A maioria das plataformas de armazenamento em nuvem favorecem disponibilidade sobre consistência com base no fato de que é difícil manter consistência forte sobre os dados que podem eventualmente ser distribuídos entre diversas e distantes localizações geográficas. Com a habilidade de controlar a localização dos dados, o custo de manter a consistência ao longo de um grupo de objetos pode ser naturalmente reduzido.

5.3 Arquitetura do sistema

A arquitetura do sistema Phoenix segue a abordagem tradicional de três camadas adotada por sistemas de banco de dados relacional, composta pelos níveis físico, lógico e externo. Os níveis lógico e externo baseiam-se nos conceitos tradicionais, mas o de armazenamento físico é fornecido por uma nuvem de armazenamento de dados. De forma abstrata, a arquitetura do sistema, apresentada na figura 5.2, é composta por três componentes: a aplicação cliente (nível externo), o banco de dados relacional (nível lógico) e repositório de dados (nível físico). No contexto deste trabalho o componente principal da arquitetura é o módulo de armazenamento, uma vez que aplicativos existentes são usados para compor as demais camadas e montar o sistema.

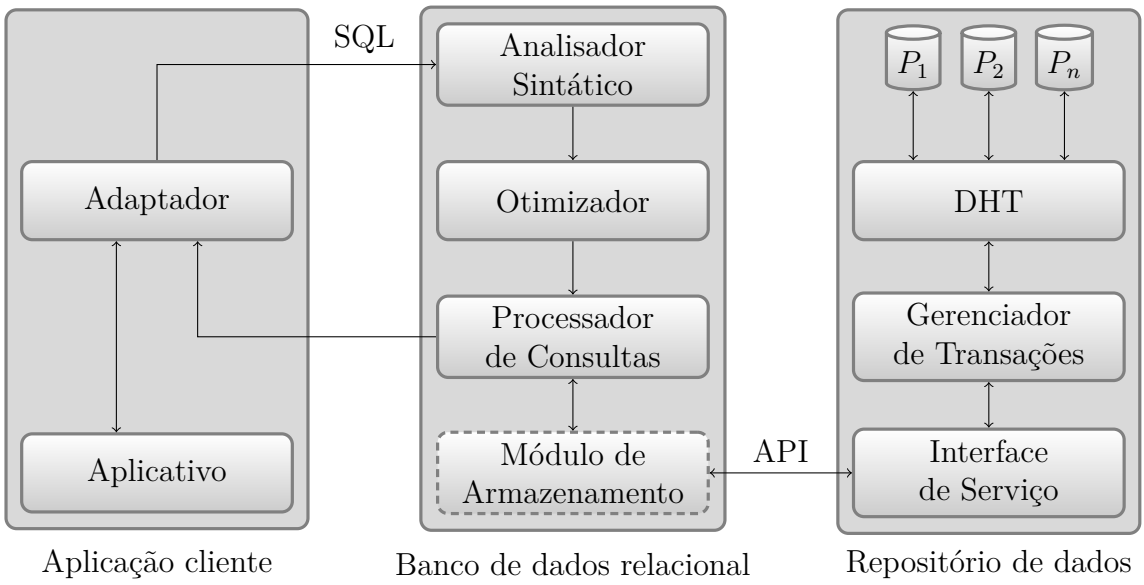

Figura 5.2: Modelo conceitual do sistema e principais componentes.

Na implementação do sistema, a camada lógica consiste no banco de dados MySQL [Oracle, 2010], enquanto que o Scalaris [Berlin and onScale solutions GmbH, 2010] atua como a camada de armazenamento em nuvem. O MySQL foi escolhido para compor o sistema devido a sua arquitetura plugável de módulo de armazenamento (*PSEA, pluggable storage engine architecture*). O módulo de armazenamento, ou camada de mapeamento, é o componente subjacente do SGBD responsável por estruturar o modelo de dados lógico do banco de dados relacional em um modelo físico, que define formas de armazenar, manipular e recuperar dados estruturados na forma de tabelas (relações). Este módulo é encarregado de prover os recursos necessários para o armazenamento e manipulação das estruturas de dados em uma forma passível de processamento pelo SGBD relacional.

O módulo de armazenamento é embutido dentro da arquitetura de armazenamento do sistema gerenciador de banco de dados relacional MySQL e realiza o mapeamento relacional para pares chave-valor e a persistência transparente do esquema do banco de dados e seus objetos (estruturas lógicas). A combinação de várias instâncias da coleção dos componentes, banco de dados relacional e repositório de dados distribuído, formam um banco de dados distribuído, cuja distribuição é homogênea e transparente aos usuários do sistema.

O serviço de armazenamento de dados na nuvem, ou camada de persistência, é responsável por garantir a permanência das informações armazenadas no sistema, na forma de um repositório de dados distribuído, escalável e transacional. Este repositório de da-

dos disponibiliza uma interface de serviço para o armazenamento de pares (ou tuplas) de chave-valor que permite a inserção, resgate e remoção de seus elementos individualmente. Ele é usado como base para a estruturação e organização de um banco de dados relacional e não apenas para pesquisas baseadas em chaves. Este serviço é implementado pelo Scalaris [Schütt et al., 2008]. A decisão de usar o Scalaris é devido à suas capacidades que mais correspondem aos requisitos definidos na Seção 5.2, ao passo que abstrai as complexidades e detalhes de implementação. Na concepção global, a camada de replicação do Scalaris provê a disponibilidade de dados, que é fundamental para evitar perda de dados ou indisponibilidade. A camada de transação lida com a concorrência, de modo que o sistema possa garantir consistência forte, atomicidade e isolamento, embora a disponibilidade possa ser sacrificada.

O módulo de armazenamento é inteiramente construído sobre a interface padrão baseada em DHT do repositório de dados e, portanto, teoricamente permitindo que ele seja executado sobre outros repositórios que disponibilizem uma interface similar. Somado à estratégia de mapeamento de dados, essa abordagem torna o sistema, de ponta a ponta, decomponível do armazenamento de dados subjacente.

5.3.1 Módulo de Armazenamento

Um módulo (ou mecanismo) de armazenamento é uma camada na arquitetura do servidor MySQL que é responsável por abstrair a camada de dados física da camada lógica do servidor, bem como possibilitar e fornecer um conjunto de operações de baixo nível a serem utilizadas pelo gerenciador do banco de dados. Assim, o módulo é o componente responsável por executar para o banco de dados as operações de entrada e saída (I/O) de dados, bem como permitir e fazer cumprir determinados conjuntos de funcionalidades necessárias pelas aplicações que utilizam o servidor.

Conceitualmente, o módulo de armazenamento deve fornecer as quatro operações básicas de armazenamento persistente: criar, ler, atualizar e excluir (CRUD). Mais especificamente, a API a ser implementada por um módulo de armazenamento está dividida em duas seções, refletindo dois conjuntos de funcionalidades: as operações de armazenar e recuperar dados e uma interface entre o módulo de armazenamento e o otimizador de consultas. O servidor MySQL se comunica com o módulo de armazenamento através desta API.

Em um nível mais elevado, um módulo de armazenamento é responsável pelo tratamento dos diferentes tipos de tabela. Cada módulo de armazenamento é uma classe cujas instâncias se comunicam com o servidor MySQL através de uma classe chamada de *handler*, que é análoga a um cursor. *Handlers* são instanciados para cada processo que necessite trabalhar com uma tabela específica. Se três sessões começarem a trabalhar com uma mesma tabela, três instâncias são criadas. A Figura 5.3 apresenta uma visão geral sobre a arquitetura do sistema de gerenciador de banco de dados MySQL.

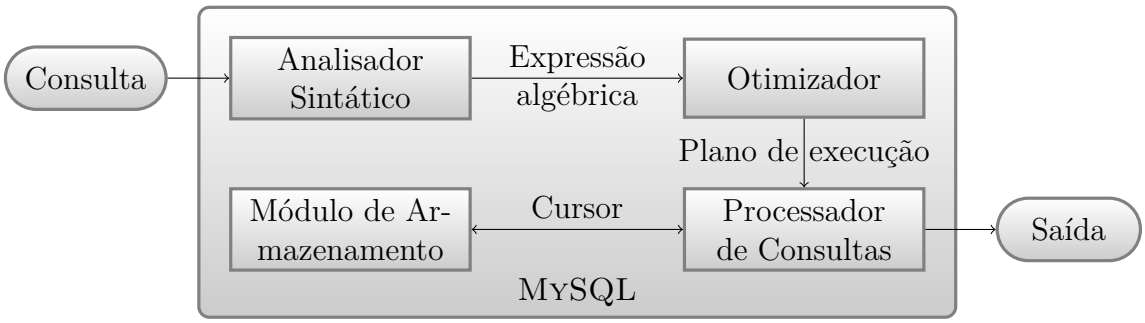

Figura 5.3: Arquitetura e principais componentes do MySQL.

Quando uma instância do *handler* é criada, o servidor MySQL envia comandos para que ela execute tarefas como a abertura e recuperação dos dados de uma tabela, manipulação de linhas e gerenciamento de índices. Módulos de armazenamento podem ser construídos de forma incremental. Assim, é possível construir um módulo de armazenamento só de leitura e posteriormente adicionar suporte para operações de inserção, atualização e remoção de linhas, e mais tarde adicionar suporte para a indexação, transações e outras operações avançadas.

Independentemente do módulo de armazenamento, cada tabela está associada a um arquivo *frm* que contém a definição da tabela com os nomes das colunas, seus tipos e tamanhos, e outras propriedades. Em essência, o arquivo *.frm* reúne e armazena as informações provenientes do comando *CREATE TABLE*. O servidor lê a tabela a partir da definição no arquivo *.frm* e armazena esta definição no *cache* de tabelas. Desta forma, na próxima vez que a tabela for acessada, não é preciso reler e analisar o arquivo *frm*.

O servidor MySQL também provê um mecanismo de exclusão mútua por tabela. Assim, cada módulo de armazenamento pode aproveitar esse recurso, ou indicar ao gerenciador de exclusão mútua para sempre conceder quaisquer requisições de bloqueio. Ou seja, é possível ignorar o núcleo de gerenciamento de exclusão mútua. Nesse caso, o módulo de

armazenamento torna-se responsável por assegurar a coerência na ocorrência de acessos concorrentes.

5.3.2 Operações básicas

No contexto do módulo de armazenamento, para recuperar e armazenar os dados é necessário estabelecer estratégias para que seja possível realizar as operações básicas de criação, consulta, atualização e remoção de dados. Estas operações são implementadas usando apenas o conjunto de primitivas *put*, *get* e *delete* que a interface do repositório de dados exporta. Essas primitivas operam sobre pares de chave-valor que, por sua vez, são compostos por objetos POEM advindos do processo de mapeamento.

Dentro do módulo de armazenamento, um componente mediador é responsável por executar as transformações necessárias. O mediador é invocado sempre que uma operação básica deve ser realizada, sendo ele o ponto de entrada do módulo de armazenamento para cada operação básica. O parâmetro de entrada das operações básicas é um *buffer* que corresponde a uma linha da tabela. Este *buffer* contém campos onde os valores dos atributos (colunas) podem ser lidos ou escritos. Conforme pode ser observado na Figura 5.4, o mediador também é responsável pela aplicação das estratégias de fragmentação, otimização e mapeamento.

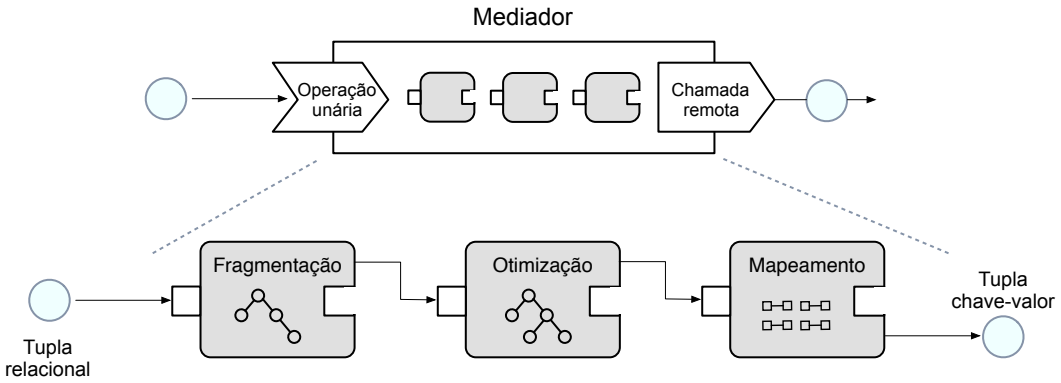

Figura 5.4: Modelo simplificado do mediador.

Para inserir uma linha em uma tabela, o mediador transforma os dados da linha em objetos POEM, que por sua vez são transformados em pares chave-valor, seguindo a estratégia delineada na seção 4.4. Por fim, os pares chave-valor são enviados para o repositório de dados distribuído através de uma chamada remota *put*. Para recuperar uma ou mais linhas da tabela, é necessário reconstruir as chaves de armazenamento.

A chave é reconstruída seguindo o mesmo processo usado para criar as chaves durante a inserção. De posse da chave de armazenamento, para recuperar o valor de armazenamento basta realizar uma chamada remota *get* usando a chave como parâmetro. Mas, para obter os dados originais, é preciso aplicar o processo inverso de mapeamento sobre o valor de armazenamento. Para remover uma ou mais linhas da tabela, basta seguir o mesmo processo de criação das chaves e realizar chamadas remotas para os respectivos procedimentos. A atualização de uma ou mais colunas de uma linha pode ser alcançada com a combinação das operações anteriores: a linha a ser alterada é buscada, removida e um nova linha contendo os novos valores é inserida.

As formas de acesso (*access path*) primordiais implementadas pelo módulo de armazenamento são a busca por chave (primária e única), varredura em ordem lexicográfica dos valores das chaves ou índices e varredura completa (*full table scan*). Devido à maneira como os dados são fragmentados e distribuídos, a forma de acesso mais eficiente é a busca por chave primária, uma vez que essa busca envolve o menor número de troca de mensagens (chamadas remotas). As formas de acesso restantes se baseiam na capacidade do Scalaris em preservar a ordem lexicográfica dos dados, e a eficiência das mesmas está atrelada a quantidade de dados que precisam ser acessados.

5.3.3 Otimização e indexação

O módulo de armazenamento pode adicionar suporte a índices através do fornecimento de informações sobre os índices ao otimizador de consultas. As informações fornecidas permitem que o otimizador tome decisões sobre quais e quando os índices devem ser usados, além de invocar os métodos de varredura implementados pelo módulo de armazenamento. Um aspecto importante da interface entre o módulo de armazenamento e otimizador é a possibilidade, por parte do módulo de armazenamento, de realizar refinamentos (melhorias) no plano de execução. O refinamento do plano é uma das últimas etapas do processo de otimização e permite ao módulo de armazenamento tomar decisões informadas sobre o que e como os registros devem ser recuperados e transformados.

Um elemento chave da etapa de refinamento é possibilidade de trazer (*push down*) para o módulo de armazenamento os predicados (condições) de busca e junção (*union*). Por exemplo, a posse dos predicados permite ao módulo de armazenamento paralelizar os processos de requisição, avaliação e combinação dos registros, além de eliminar a necessidade

de recuperar e processar registros irrelevantes. Esta técnica pode melhorar drasticamente o desempenho de módulos que dependem de armazenamento baseado em rede, reduzindo a quantidade de mensagens e dados que precisam ser trocados e recuperados.

Assim, no contexto deste trabalho, o uso de índices deve permitir uma abordagem de acesso mínimo ao repositório de dados para consultas por abrangência que utilizam operadores relacionais, a fim de reduzir o tráfego e, por conseguinte, mitigar atrasos (e.g. latência) causados pela rede. A maneira básica de dar suporte a consultas por abrangência é a de se apoiar na capacidade do Scalaris em armazenar os dados em ordem lexicográfica e de selecionar (recuperar) pares de chave-valor dentro de um dado intervalo lexicográfico através da operação *range_read*.

Haja vista a capacidade de controlar a localização do conteúdo, conforme descrito na subseção 5.2.1, é possível realizar consultas por abrangência sobre expressões de caminho similares. Dado um conjunto de pares chave-valor $L = \{(/livros/livro/título/1,$ "$Iracema$"$)$, $(/livros/livro/título/2,$ "$O\ Seminarista$"$)$, ..., $(/livros/livro/título/id_n$ "$Título_n$"$)\}$, uma função *chave* que retorna o valor da chave de armazenamento e uma consulta por abrangência representada pela expressão em álgebra relacional $\sigma_{I(x,y)}(L)$, que recupera todos os pares $l_i \in L$ tal que $x \leq chave(l_i) \leq y$ segundo a ordenação lexicográfica em Q. Um exemplo de consulta seria $\sigma_{I(/livros/livro/título/0,/livros/livro/título/10)}(L)$ que seleciona os títulos dos livros com identificadores entre 0 e 10.

5.4 Limitações

Este trabalho não contempla algumas propriedades, relacionamentos e restrições que normalmente fazem parte de um sistema gerenciador de banco de dados completo. Inicialmente, regras de integridade referencial não são aplicadas. Ademais, os planos de execução não são modificados para um melhor desempenho, ou seja, não ocorre qualquer tipo de otimização sobre as possíveis formas de execução de uma consulta, exceto pela utilização de índices através da manutenção de chaves em ordem lexicográfica.

No próximo capítulo são apresentados os experimentos realizados para validar o sistema proposto e para verificar a influência da estratégia de integração e distribuição de dados. Os aspectos de escalabilidade, fragmentação e mapeamento levantados neste trabalho são analisados mediante a realização de experimentos específicos.

CAPÍTULO 6

EXPERIMENTOS

Nesta capítulo, são apresentados os resultados dos experimentos destinados a determinar a viabilidade e escalabilidade da arquitetura proposta. Entre os objetivos dos experimentos estão medir o custo adicional imposto pela arquitetura do sistema, de modo a delinear o impacto da estrutura baseada em camadas sobre o desempenho do sistema de armazenamento em nuvem, e compreender o impacto das estratégias de fragmentação e mapeamento sobre a taxa de processamento do sistema.

A arquitetura introduzida no capítulo 5 é validada através de dois tipos principais de experimentos: *benchmark* de busca e atualização e processamento transacional. O experimento de busca e atualização visa validar os requerimentos de processamento de consultas, escalabilidade e medir a sobrecarga imposta pela estratégia de integração e distribuição de dados. Os experimentos de processamento transacional buscam simular uma carga de trabalho de um sistema de processamento on-line de transações (OLTP) a fim de avaliar o comportamento do sistema e como a estratégia de fragmentação pode afetá-lo.

6.1 Ambiente

O experimento foi realizado em um conjunto de 6 servidores genéricos interligados através de uma LAN departamental. Cada servidor possui um processador *dual core* com 2.40GHz, 2 GB de RAM e placa ethernet de 100Mbps. Os servidores não possuem disco rígido, sendo assim, as bases de dados dos experimentos foram armazenados em memória. As diferentes instâncias dos processos do MySQL, Scalaris e demais ferramentas são distribuídas igualmente pelos servidores. Assim, os servidores desempenham vários papéis no ambiente cliente/servidor de uma rede.

A comunicação do módulo de armazenamento para com o Scalaris é através de chamadas de procedimento remoto codificadas em JSON (similar ao XML-RPC), utilizando transporte HTTP, enquanto que a comunicação entre os nós do Scalaris segue o mecanismo nativo de comunicação do Erlang (plataforma na qual o Scalaris foi desenvolvido).

O Scalaris é configurado com um fator de replicação dois, ou seja, o sistema mantém duas réplicas de cada item de dados armazenado.

6.2 Sobrecarga e escalabilidade

O mecanismo do experimento de busca e atualização consiste em uma transação que recupera e atualiza um registro de uma tabela. A tabela é composta por duas colunas do tipo inteiro, sendo que a primeira é uma chave primária. Nesta primeira coluna é armazenado o identificador único de cada cliente, enquanto que a segunda coluna é um contador referente à quantidade de vezes que o cliente executou a transação. Assim, para executar esta transação, o servidor de banco de dados precisa recuperar, através da DHT, o valor do contador associado ao identificador do cliente, incrementá-lo e atualizá-lo. Com base em um esquema de fragmentação horizontal, a transação se traduz em uma operação de leitura que recupera um único registro, e uma operação de escrita que atualiza este registro.

A fim de medir a sobrecarga imposta pela camada de integração e distribuição, a transação é programada de maneiras diferentes de modo que uma variação se comunique através da camada de integração e distribuição, e outra que se comunica diretamente com a DHT. A primeira variação executa um único comando SQL para incrementar o contador (UPDATE tabela SET contador = contador+1 WHERE id = *chave*). A segunda variação realiza chamadas remotas de procedimento diretamente à DHT, utilizando as operações de leitura e escrita (*put*/*get*) do Scalaris. Note que ambas as variações implicam num mesmo número e tipo de operações da DHT.

É válido lembrar que como cada cliente possui um identificar único e global, não existe conflito entre as transações. Na prática, esse caso ideal raramente é alcançado devido a conflitos de concorrência entres transações que acessam um mesmo subconjunto de registros. Mas, para observar a evolução do sistema e o efeito preciso da camada sobre a taxa de transações, este cenário se faz necessário.

6.2.1 Experimento

O experimento é executado em conjuntos de dois e quatro servidores, a fim de determinar a escalabilidade horizontal. Cada cliente executa 3.000 iterações da transação. O número

de clientes simultâneos varia de 16 à 128, em incrementos de 16. Os clientes, bem como as instâncias do servidor MySQL, são distribuídos de maneira proporcional entre os servidores. Ou seja, na rodada com 2 servidores e 80 clientes simultâneos, em cada servidor existem 40 clientes locais que se comunicam com a instância MySQL e/ou nó DHT sendo executado no mesmo servidor.

A unidade de medida básica do experimento é o número de transações que são completadas por todos os clientes dentro do período de medição. Em cada execução de um experimento (rodada) é coletada a taxa de transações por segundo (TPS) e o tempo médio de resposta por transação (TRT) em segundos. Dado um número de clientes, a taxa de transações por segundo fornece uma medida do número de operações que o sistema é capaz de processar em um segundo. O tempo médio de reposta por transação determina o quão rápido o sistema processa uma transação, considerando também o número de clientes. Para fins de cálculo do TRT é considerado apenas o tempo decorrido entre o início e a conclusão da transação.

Os resultados do experimento executado com 2 servidores são apresentados na Tabela 6.1, enquanto a Tabela 6.2 mostra os resultados considerando 4 servidores. Para cada rodada com um número específico de clientes simultâneos, ainda é mostrado o tempo total, em segundos, e a diferença em percentagem da taxa de transações por segundo das duas variações.

Clientes	Phoenix			Scalaris			Dif. TPS %
	Total (s)	TPS	TRT	Total (s)	TPS	TRT	
16	6	16184	2.479×10^{-4}	5	17725	2.271×10^{-4}	-8.69
32	25	7608	5.258×10^{-4}	23	8357	4.804×10^{-4}	-8.97
48	59	4900	8.171×10^{-4}	58	4951	8.097×10^{-4}	-1.02
64	115	3340	1.198×10^{-3}	95	4069	9.872×10^{-4}	-17.90
80	184	2608	1.534×10^{-3}	167	2876	1.392×10^{-3}	-9.31
96	270	2134	1.877×10^{-3}	262	2202	1.820×10^{-3}	-3.08
112	384	1750	2.287×10^{-3}	351	1916	2.090×10^{-3}	-8.63
128	527	1460	2.743×10^{-3}	526	1461	2.738×10^{-3}	-0.06

Tabela 6.1: Experimento com 2 servidores.

6.2.2 Análise

Observando as primeiras linhas da Tabela 6.2, a primeira conclusão que podemos extrair é a de que o desempenho do Scalaris sofre uma degradação acentuada, em torno de 50%,

Clientes	Phoenix			Scalaris			Dif. TPS %
	Total (s)	TPS	TRT	Total (s)	TPS	TRT	
16	3	76029	2.167×10^{-4}	3	78290	2.108×10^{-4}	-2.89
32	10	37904	4.237×10^{-4}	11	36659	4.417×10^{-4}	3.40
48	27	21836	7.397×10^{-4}	25	22808	7.019×10^{-4}	-4.26
64	51	15114	1.061×10^{-3}	47	16358	9.823×10^{-4}	-7.60
80	79	12158	1.319×10^{-3}	75	12785	1.253×10^{-3}	-4.90
96	114	10180	1.577×10^{-3}	107	10771	1.489×10^{-3}	-5.48
112	164	8194	1.957×10^{-3}	154	8764	1.835×10^{-3}	-6.51
128	225	6860	2.339×10^{-3}	214	7192	2.227×10^{-3}	-4.61

Tabela 6.2: Experimento com 4 servidores.

quando o número de cliente simultâneos salta de 16 para 32. Após 48 clientes, e ao longo do experimento, a degradação vai atingindo patamares menores. Apesar desta queda abrupta, o tempo médio de resposta por transação sofre uma degradação praticamente linear, em torno de 3×10^{-3} segundo a cada rodada.

Considerando a diferença percentual dos valores da taxa de transação por segundo das variações, é possível afirmar que a sobrecarga associada à camada de integração e distribuição do Phoenix é algo em torno de 1% a 10%. Em termos do tempo médio de resposta, a diferença é praticamente irrisória.

Estabelecendo um paralelo entre os números de transações por segundo do experimento com 2 servidores ao de 4 servidores, é possível afirmar que a adição de 2 servidores praticamente quadruplicou o desempenho do sistema. Por outro lado, o tempo médio de reposta praticamente não se alterou. Ambos os casos se devem a forma como os dados são distribuídos na DHT. Como as chaves são distribuídas uniformemente entre os nós da DHT, a adição de nós ocasiona uma expansão na capacidade total do sistema, mas não muda o cenário de processamento de cada transação individual. Assim, demonstra-se que o sistema é escalável horizontalmente, haja vista que a adição de novos servidores proporcionou um aumento do processamento de transações.

6.3 Processamento de transações

Sistemas OLTP são caracterizados por transações muito breves, mas muito freqüentes, que possuem poucas ou nenhumas operações de agregação e que envolvem pequenos grupos de linhas. Exemplos clássicos de OLTP são sistemas financeiros e de vendas no varejo. A natureza dos ambientes OLTP é predominantemente de uso interativo com tempo de

resposta curto, que envolvam transações pequenas, mas de alta concorrência. Sistemas OLTP requerem tempos de resposta curtos para que os usuários possam permanecer produtivos. Devido à grande quantidade de usuários, curtos tempos de resposta, e as transações de pequeno porte, a simultaneidade em ambientes OLTP é muito alta.

Para realizar este experimento, foram feitas algumas tentativas de utilizar as *benchmarks* TPC-E [Council, 2010b] e/ou TPC-C [Council, 2010a], que são as cargas de trabalho OLTP com uma mistura de operações de somente leitura e atualização intensivas. O TPC-E modela a atividade de uma empresa corretora de valores que precisa gerenciar contas de clientes, executar ordens de compra e venda, e ser responsável pela interações dos clientes com os mercados financeiros. Já o TPC-C, simula o ambiente de shopping on-line, no qual os usuários executam transações de compra, entrada e entrega de pedidos, registro de pagamentos e acompanhamento do nível de estoque.

Infelizmente, as implementações destas *benchmarks* não se adéquam ao ambiente e/ou sistema. Por exemplo, a carga de trabalho do TCP-E necessita de suporte a chave estrangeira, uma funcionalidade que, por enquanto, não é contemplada neste trabalho. O TPC-C, por outro lado, não se adéqua ao ambiente de experimentos devido ao grande volume de dados necessário para executar a versão mais simples da *benchmark*. Com um fator de escala 1, o TPC-C gera uma carga de dados em torno de 250 MB de dados, o que acaba resultando em uma carga total de 1000 MB por servidor em razão do fator de replicação. Devido a estes problemas, a solução encontrada foi utilizar uma *benchmark* alternativa chamada SysBench [MySQL AB, 2010a].

6.3.1 SysBench

O SysBench é uma coleção de programas (módulos) de *benchmark* destinados a avaliar o desempenho de cargas de trabalho relacionadas a OLTP. Os módulos existentes permitem avaliar diversos parâmetros de um sistema operacional (escalonador, sistema de arquivo, etc.), bem como o desempenho de um servidor de banco de dados. A arquitetura do Sys-Bensh é relativamente simples, sendo basicamente composta por dois módulos de função: controlador e cliente. Os clientes são responsáveis pela criação das requisições especificadas pelo usuário, enquanto que o controlador controla o início e o fim da *benchmark*. Os clientes são implementados usando *threads* para executar as requisições e se comunicam com um servidor MySQL usando *sockets*.

No módulo OLTP destinado a avaliar o desempenho de bancos de dados, as requisições podem ser de três tipos: transacional complexa, transacional simples e não-transacional. O tipo transacional complexo inclui, em um contexto transacional, diversos tipos de consultas (de igualdade, abrangência, etc.), atualizações, inserções e exclusões. As requisições do tipo transacional simples e não-transacional contem um subconjunto da transacional complexa. Uma requisição transacional complexa é composta de 21 operações, sendo que a última operação é uma confirmação (*commit*) da transação iniciada para executar a requisição.

Para qualquer um dos tipos de requisição, o SysBench utiliza uma tabela de tamanho predeterminado. Esta tabela é composta por quatro colunas, sendo que as duas primeiras são do tipo inteiro e as restantes são do tipo seqüência de caracteres de comprimento fixo. A primeira coluna faz parte da chave primária da tabela. As *threads* cliente executam transações sobre esta tabela, sendo que cada transação pode conter uma série de comandos, de acordo com o tipo de requisição especificado pelo usuário. A lista completa dos tipos de requisição, bem como o esquema detalhado da tabela, estão disponíveis em [MySQL AB, 2010b].

6.3.2 Experimento

O tipo de requisição selecionada é a transacional complexa. Leitura e escrita são permitidas em uma única transação, sendo que 66% das operações da transação são de leitura, enquanto que 23% são de escrita. Todas as transações inserem, excluem, atualizam e selecionam na tabela única durante um período de 300 segundos. Apesar do intervalo de medição ser baixo, provas por amostragem com intervalos de duração de até 20 minutos não demonstraram diferenças relevantes. A fim de obter resultados estáveis, é realizado um aquecimento rápido antes de cada rodada. O tamanho da tabela é de 100 mil linhas.

A DHT é espalhada por quatro servidores, sendo que em um deles é executado o SysBench e o banco de dados MySQL. Os testes foram executados variando o número de clientes (*threads*) que acessam o banco de dados, de 1 a 128 clientes simultâneos, com incrementos de 16. A medida básica de desempenho é a taxa de transações durante o período de tempo supracitado. A escalabilidade é medida pela variação da taxa de transações quando da variação do número de clientes. Também é coletado o número total de operações de leitura e escrita e os tempos de execução mínimo, médio e máximo de

todas as requisições completadas durante uma rodada.

Os resultados do experimento são apresentados na tabela Tabela 6.3 e mostram o número máximo de transações por segundo que o sistema suporta, dependendo do número de clientes (primeira coluna). Para cada rodada específica, a tabela também mostra os números totais, incluindo a taxa por segundo, das operações de leitura e escrita executadas, os tempos de execução em milissegundos, e, por último, o percentil 95 dos tempos de execução. A Figura 6.1 apresenta a escalabilidade considerando os valores de TPS quando da variação do número de clientes.

Thrs.	TPS	Leitura	Escrita	L.E./s	Min.	Méd.	Máx.	95%
1	213.06	894852	319590	4048.11	3.22	4.69	52.57	9.96
2	380.50	1598100	570750	7229.47	3.85	5.25	51.14	6.13
4	537.56	2257780	806350	10213.61	4.87	7.43	71.31	9.34
8	600.25	2521246	900437	11404.88	3.75	13.32	1817.00	17.88
16	595.47	2501100	893250	11314.02	10.42	26.86	306.18	37.97
32	560.34	2353652	840583	10646.58	20.68	57.09	1830.21	79.54
64	506.88	2129442	760497	9631.15	4.15	126.24	1960.06	159.19
128	76.09	320040	114297	1445.71	446.70	1681.76	11633.69	1935.99

Tabela 6.3: SysBench: OLTP transacional complexo.

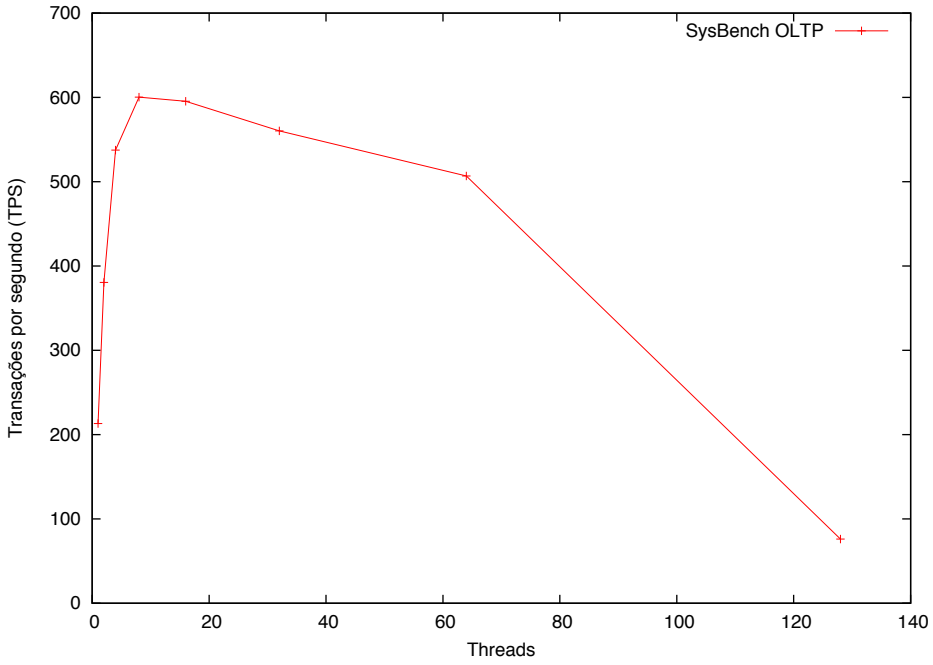

Figura 6.1: Escalabilidade segundo o número de clientes.

6.3.3 Análise

Os resultados do experimento mostram que um número alto de clientes pode impactar negativamente a taxa de transações e os tempos de resposta do sistema. Observando os dados da tabela, é perceptível uma queda acentuada do desempenho após 64 clientes simultâneos. A 128 clientes, o tempo de espera por uma transação dura em média 1.9 segundos, podendo chegar até a 11 segundos. Essa queda acentuada pode ser explicada pelo fraco poder computacional dos servidores do ambiente, que contam com apenas dois *cores* de processamento. Essa competição por recursos (tempo de CPU) faz com que surjam problemas de bloqueio, o que acaba prejudicando demasiadamente o desempenho do sistema. Este comportamento também pode explicar o pico de tempo máximo com 8 clientes. Enfim, nesses casos se faz necessário limitar o número máximo de clientes simultâneos.

Ademais, é possível afirmar que o sistema é capaz de sustentar uma taxa estável de transações até para 64 clientes simultâneos, se desconsiderado requisitos de tempo de resposta. O ponto ideal entre taxa de transações e tempo de resposta está entre 4 e 8 clientes simultâneos. De 8 a 64 clientes as taxas de operações são similares, mas ocorre acentuada degradação dos tempos de execução.

6.4 Fragmentação e localização

Uma técnica freqüentemente utilizada para aprimorar o desempenhado de bancos de dados distribuídos OLTP é a fragmentação de dados. No contexto deste sistema, a fragmentação aprimora a concorrência devido a uma potencial redução na necessidade de comunicação e coordenação entre o banco de dados e a DHT para acessar os dados. Conforme abordado nas seções 3.1 e 3.2, uma estratégia ideal de fragmentação visa propiciar que as transações somente acessem fragmentos relevantes, e, potencialmente, precisem acessar somente um único fragmento.

A partir de uma análise da carga de trabalho OLTP do SysBench, é possível identificar uma estratégia de fragmentação horizontal que proporcione uma melhor difusão dos dados. Conforme explicado na seção anterior, a carga de trabalho do SysBench é predominantemente de leitura e composta por algumas consultas por abrangência. A freqüência destas consultas por abrangência é maior do que a de consultas por igualdade. Uma es-

tratégia de fragmentação horizontal que fragmente os dados de forma que as consultas por abrangência, e a transações de modo geral, possam ser executas em um número pequeno (ou único) de nós, pode ter um impacto no desempenho.

6.4.1 Experimento

Dentro dos parâmetros de funcionamento desta carga de trabalho em modo somente leitura, podemos controlar um conjunto de fatores para averiguar essa hipótese. Para tanto, é utilizado uma configuração da DHT com 6 nós, cada qual em um servidor diferente. O tamanho da tabela é de 3000 linhas e o intervalo das consultas por abrangência é de 200 linhas. Para fins de comparação, a relação (tabela) é inicialmente dividida horizontalmente em 12 fragmentos, depois em 18 e 24 fragmentos. No primeiro esquema de fragmentação, em cada nó haverá dois fragmentos, cada qual equivalente a 250 linhas da tabela. No segundo, cada fragmento equivale a aproximadamente 180 linhas e a 125 linhas no terceiro.

O experimento é realizado desta forma para propiciar que as consultas por abrangência necessitem acessar somente um único nó no primeiro esquema de fragmentação, dois nós no segundo e assim em diante. Desta forma e com base no fator de replicação, cada nó é responsável pelo número de fragmentos dividido pelo número total de nós, mais o número equivalente de réplicas. Assim, por exemplo, no experimento com 12 fragmentos cada nó é responsável por 2 fragmentos e por 2 réplicas. A difusão dos fragmentos e consultas é uniforme e homogênea.

Este experimento segue os mesmos parâmetros do experimento anterior, exceto o número de clientes. O número de clientes simultâneos varia entre 2 e 10, com incrementos de 2. Os resultados são apresentados na tabela 6.4 e mostram o número de transações por segundo e o número total de operações de leitura. A última coluna dos campos de 16 e 24 fragmentos representa a diferença em percentagem da taxa de transações por segundo, em relação ao experimento de 12 fragmentos.

6.4.2 Análise

As métricas tabuladas corroboram a noção que a circunstância favorável pela fragmentação é quando a transação necessita acessar um número menor de fragmentos. Neste

Cli.	Fragmentos: 12		Fragmentos: 16			Fragmentos: 24		
	TPS	Leitura	TPS	Leitura	%	TPS	Leitura	%
2	421.37	1769768	415.86	1746654	1.32	402.10	1688834	4.79
4	606.89	2548980	589.91	2477664	2.88	579.55	2434138	4.72
6	699.98	2939986	666.12	2797774	5.08	657.40	2761136	6.48
8	715.86	3006668	669.63	2812488	6.90	658.29	2764874	8.75
10	696.39	2924964	652.72	2741508	6.69	643.62	2703288	8.20

Tabela 6.4: Experimento de fragmentação e localização.

cenário, os resultados colocam o custo de acessar um segundo fragmento em torno de 1.3% a 6.7%, dependendo do número clientes. Acessar um terceiro fragmento aumenta este custo em torno 1.5%, de acordo com o número de clientes. A diferença entre as porcentagens de perda de desempenho pode ser explicada pelos custos de comunicação.

Num sistema distribuído, o tempo de comunicação é basicamente composto por dois elementos: o tempo de transmissão e a latência [Johansson, 2000]. A latência é um custo fixo e inicial inerente a cada mensagem, enquanto que o tempo de transmissão é proporcional à quantidade de dados a serem transmitidos. Como neste cenário o volume de dados a ser transmitido é basicamente constante, a desvantagem inerente ao aumento do número de nós é a latência.

Esta diferença salienta um *trade-off* importante. Haja vista que a granularidade da fragmentação está associada ao custo de comunicação, independente da velocidade da rede, requisições de menor granularidade em relação aos fragmentos possuem um tempo de resposta melhor do que requisições de maior granularidade. No entanto, o uso de fragmentos de maior granularidade implica uma redução do número total de fragmentos no sistema e, por conseguinte, uma redução na quantidade de troca de mensagens. Fragmentos de menor granularidade implicam um volume maior de fragmentos, mas exigem um número maior de mensagens necessárias a readquirir a mesma quantidade de informações.

Durante o experimento este efeito pode ser observado na distribuição de freqüência relativa à quantidade de operações de leitura realizadas em cada nó. Enquanto o experimento com 12 fragmentos tem uma distribuição assimétrica, indicando uma menor troca de mensagens entre os nós, os experimentos com 16 e 24 fragmentos obtiveram uma distribuição aproximadamente simétrica, indicando uma maior troca de mensagens e, por conseguinte, um maior custo inerente de comunicação.

CAPÍTULO 7

CONCLUSÕES E TRABALHOS FUTUROS

Neste trabalho, propõe-se um sistema de gestão de dados para modelagem, indexação e resgate de dados relacionais armazenados na forma de um repositório de dados distribuídos e hospedados em diversos nós maciçamente distribuídos, autônomos e móveis. A organização e arquitetura do sistema seguem uma abordagem estratificada, cujo alicerce é um repositório de dados distribuído baseado em tabelas de dispersão distribuídas. Dentre os desafios mais significativos do projeto estão o mapeamento entre o modelo relacional do banco de dados e o modelo semi-estruturado do repositório de dados, e a análise do desempenho da estratégia de mapeamento.

A motivação preponderante no desenvolvimento deste sistema é fornecer um serviço escalável de banco de dados relacional, sem sacrificar as principais características encontradas no tradicional banco de dados relacional, tais como transações e independência de dados, sem incorrer em complexidades desnecessárias. Embora uma abordagem comum em trabalhos anteriores para aumentar a escalabilidade do sistema depende de uma noção fraca de consistência, os experimentos demonstram que é possível alcançar escalabilidade sem sacrificar consistência. A vantagem central da composição dessas tecnologias em uma arquitetura global é uma redução nos custos de desenvolvimento, implantação e manutenção, quando comparado a banco de dados distribuídos tradicionais.

Há uma série de questões a serem investigas no futuro. Uma delas é explorar a interface entre o módulo de armazenamento e o otimizador de consultas. Por exemplo, a técnica de refinamento pode ser aplicada para aumentar o desempenho de junções de uma ou mais tabelas relacionais, de forma a reduzir a quantidade de dados que precisam ser recuperados e processados.

Outras questões que merecem uma análise mais aprofundada incluem: o armazenamento de metadados no serviço de armazenamento de dados da nuvem, incluindo permissões de acesso, a fragmentação de dados e geração de chaves de apoio para restrições, como chaves estrangeiras; geração automática de esquemas de fragmentação de dados e definição de uma linguagem de alto nível, como o SQL, para permitir que os esquemas de

fragmentação sejam definidos pelo usuário.

Com base no POEM, também foi definida uma estratégia geral de mapeamento de dados e operações entre a camada física e lógica da arquitetura proposta. Embora neste trabalho o nível lógico considerado constasse apenas o modelo relacional, a técnica de mapeamento pode ser estendida para suportar outros modelos de dados, como XML.

BIBLIOGRAFIA

[Abadi, 2009] Abadi, D. J. (2009). Data management in the cloud: Limitations and opportunities. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 32(1):3–12.

[Amazon, 2007] Amazon (2007). Amazon SimpleDB. `http://aws.amazon.com/simpledb/`.

[Androutsellis-Theotokis and Spinellis, 2004] Androutsellis-Theotokis, S. and Spinellis, D. (2004). A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys*, 36(4):335–371.

[Armbrust et al., 2009] Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D. A., Rabkin, A., Stoica, I., and Zaharia, M. (2009). Above the clouds: A berkeley view of cloud computing.

[Aspnes and Shah, 2007] Aspnes, J. and Shah, G. (2007). Skip graphs. *ACM Transactions on Algorithms*, 3(4):37.

[Berlin and onScale solutions GmbH, 2010] Berlin, Z. I. and onScale solutions GmbH (2010). Scalaris - distributed transactional key-value store.

[Bernstein et al., 2002] Bernstein, P., Giunchiglia, F., Kementsietsidis, A., Mylopoulos, J., Serafini, L., and Zaihrayeu, I. (2002). Data management for peer-to-peer computing: A vision.

[Bharambe et al., 2004] Bharambe, A. R., Agrawal, M., and Seshan, S. (2004). Mercury: supporting scalable multi-attribute range queries. *SIGCOMM Comput. Commun. Rev.*, 34(4):353–366.

[Brewer, 2000] Brewer, E. A. (2000). Towards robust distributed systems. In *PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, pages 7+, New York, NY, USA. ACM.

[Buyya et al., 2008] Buyya, R., Yeo, C. S., and Venugopal, S. (2008). Market-oriented Cloud Computing: Vision, Hype, and Reality for Delivering IT Services as Computing Utilities. In *Proceedings of the 10th IEEE International Conference on High Performance Computing and Communications (HPCC 2008)*, pages 5–13, Dalian, China. IEEE Computer Society: Los Alamitos, CA, USA.

[Cattell, 2010] Cattell, R. (2010). High performance scalable data stores. available at `http://cattell.net/datastores/Datastores.pdf/`.

[Ceri et al., 1982] Ceri, S., Negri, M., and Pelagatti, G. (1982). Horizontal data partitioning in database design. In *SIGMOD '82: Proceedings of the 1982 ACM SIGMOD international conference on Management of data*, pages 128–136, New York, NY, USA. ACM.

[Chang et al., 2006] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. (2006). Bigtable: a distributed storage system for structured data. In *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, page 15, Berkeley, CA, USA. USENIX Association.

[Chawathe et al., 2005] Chawathe, Y., Ramabhadran, S., Ratnasamy, S., LaMarca, A., Shenker, S., and Hellerstein, J. M. (2005). A case study in building layered dht applications. In GuÃ©rin, R., Govindan, R., and Minshall, G., editors, *SIGCOMM*, pages 97–108. ACM.

[Cooper et al., 2008] Cooper, B. F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H. A., Puz, N., Weaver, D., and Yerneni, R. (2008). Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288.

[Council, 2010a] Council, T. P. P. (2010a). TPC-C. `http://www.tpc.org/tpcc/`.

[Council, 2010b] Council, T. P. P. (2010b). TPC-E. `http://www.tpc.org/tpce/`.

[Curino et al., 2010] Curino, C., Jones, E., Zhang, Y., Wu, E., and Madden, S. (2010). Relational cloud: The case for a database service. Technical Report MIT-CSAIL-TR-2010-014, Massachussets Institute of Technology.

[Das et al., 2010] Das, S., Agrawal, D., and Abbadi, A. E. (2010). G-store: A scalable data store for transactional multi key access in the cloud. In *ACM SOCC*, Indianapolis, IN. ACM, ACM.

[Datta et al., 2005] Datta, A., Hauswirth, M., John, R., Schmidt, R., and Aberer, K. (2005). Range queries in trie-structured overlays. In *P2P'05: Proceedings of the 5th International Conference on Peer-to-Peer Computing*.

[DeCandia et al., 2007] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. (2007). Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220.

[Egger, 2009] Egger, D. (2009). Sql in the cloud. Master's thesis, Swiss Federal Institute of Technology Zurich (ETH).

[Gançarski et al., 2002] Gançarski, S., Naacke, H., Pacitti, E., and Valduriez, P. (2002). Parallel processing with autonomous databases in a cluster system. In *On the Move to Meaningful Internet Systems, 2002 - DOA/CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002*, pages 410–428, London, UK. Springer-Verlag.

[Gao and Steenkiste, 2004] Gao, J. and Steenkiste, P. (2004). An adaptive protocol for efficient support of range queries in dht-based systems. In *ICNP '04: Proceedings of the 12th IEEE International Conference on Network Protocols*, pages 239–250, Washington, DC, USA. IEEE Computer Society.

[Ghemawat et al., 2003] Ghemawat, S., Gobioff, H., and Leung, S. T. (2003). The google file system. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, volume 37, pages 29–43, New York, NY, USA. ACM.

[Ghodsi et al., 2005] Ghodsi, A., Alima, L. O., and Haridi, S. (2005). Symmetric replication for structured peer-to-peer systems. In Moro, G., Bergamaschi, S., Joseph, S., Morin, J.-H., and Ouksel, A. M., editors, *Proceedings of The 3rd International Workshop on Databases, Information Systems and Peer-to-Peer Computing*, volume 4125 of *Lecture Notes in Computer Science*, pages 74–85, Trondheim, Norway. Springer.

[Google, 2008] Google (2008). App engine datastore.

[Gray and Lamport, 2006] Gray, J. and Lamport, L. (2006). Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160.

[Gribble et al., 2001] Gribble, S., Halevy, A., Ives, Z., Rodrig, M., and Suciu, D. (2001). What can databases do for peer-to-peer. In *WebDB Workshop on Databases and the Web*.

[Haridi and Moser, 2007] Haridi, S. and Moser, M. (2007). Atomic commitment in transactional dhts. In *Proceedings of Towards Next Generation Grids: Proceedings of the CoreGRID Symposium 2007*, page 11, Rennes, France.

[Harvey et al., 2003a] Harvey, N. J., Jones, M. B., Saroiu, S., Theimer, M., and Wolman, A. (2003a). Skipnet: A scalable overlay network with practical locality properties. In *Fourth USENIX Symposium on Internet Technologies and Systems (USITS)*.

[Harvey et al., 2003b] Harvey, N. J. A., Jones, M. B., Saroiu, S., Theimer, M., and Wolman, A. (2003b). Skipnet: a scalable overlay network with practical locality properties. In *USITS'03: Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems*, page 9, Berkeley, CA, USA. USENIX Association.

[Huebsch et al., 2005] Huebsch, R., Chun, B., Hellerstein, J. M., Loo, B. T., Maniatis, P., Roscoe, T., Shenker, S., Stoica, I., and Yumerefendi, A. R. (2005). The architecture of pier: an internet-scale query processor. In *IN CIDR*, pages 28–43.

[Huebsch et al., 2003] Huebsch, R., Hellerstein, J. M., Lanham, N., Loo, B. T., Shenker, S., Shenker, L. S., and Stoica, I. (2003). Querying the internet with pier. In *In VLDB*, pages 321–332.

[Johansson, 2000] Johansson, J. (2000). On the impact of network latency on distributed systems design. *Information Technology and Management*, 1:183–194. 10.1023/A:1019121024410.

[Karger et al., 1997] Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., and Lewin, D. (1997). Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *STOC '97: Proceedings of the*

twenty-ninth annual ACM symposium on Theory of computing, pages 654–663, New York, NY, USA. ACM.

[Keleher et al., 2002] Keleher, P., Bhattacharjee, B., and Silaghi, B. (2002). Are virtualized overlay networks too much of a good thing? In *First International Workshop on Peer-to-Peer Systems (IPTPS)*.

[Koloniari and Pitoura, 2005] Koloniari, G. and Pitoura, E. (2005). Peer-to-peer management of xml data: issues and research challenges. *ACM SIGMOD Record*, 34(2):6–17.

[Kubiatowicz et al., 2000] Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Wells, C., and Zhao, B. (2000). Oceanstore: an architecture for global-scale persistent storage. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, volume 28, pages 190–201, New York, NY, USA. ACM.

[Lakshman and Malik, 2010] Lakshman, A. and Malik, P. (2010). Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40.

[Liau et al., 2004] Liau, C. Y., Ng, W. S., Shu, Y., Tan, K.-L., and Bressan, S. (2004). Efficient range queries and fast lookup services for scalable p2p networks. In Ng, W. S., Ooi, B. C., Ouksel, A. M., and Sartori, C., editors, *DBISP2P*, volume 3367 of *Lecture Notes in Computer Science*, pages 93–106. Springer.

[Maly, 2003] Maly, R. J. (2003). Comparison of Centralized (Client-Server) and Decentralized (Peer-to-Peer) Networking. Semester Thesis.

[Microsoft, 2010] Microsoft (2010). Windows azure platform.

[MySQL AB, 2010a] MySQL AB (2010a). SysBench: a system performance benchmark. `http://sysbench.sourceforge.net/`.

[MySQL AB, 2010b] MySQL AB (2010b). SysBench manual. `http://sysbench.sourceforge.net/docs/#database_mode`.

[Navathe et al., 1984] Navathe, S., Ceri, S., Wiederhold, G., and Dou, J. (1984). Vertical partitioning algorithms for database design. *ACM Trans. Database Syst.*, 9(4):680–710.

[Navathe et al., 1995] Navathe, S. B., Karlapalem, K., and Ra, M. (1995). A mixed fragmentation methodology for initial distributed database design. *Journal of Computer and Software Engineering*, 3(4):395–426.

[Oracle, 2010] Oracle (2010). MySQL Server. `http://www.mysql.com`.

[Papakonstantinou et al., 1995] Papakonstantinou, Y., Garcia-Molina, H., and Widom, J. (1995). Object exchange across heterogeneous information sources. In *Eleventh International Conference on Data Engineering (ICDE 1995)*, Taipei, Taiwan. IEEE Computer Society. One cut-and-paste figure missing from ps FILE.

[Ramabhadran et al., 2004] Ramabhadran, S., Ratnasamy, S., Hellerstein, J. M., and Shenker, S. (2004). Prefix hash tree: An indexing data structure over distributed hash tables. In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing (PODC'04)*, pages 368–368, St. John's, Newfoundland, Canada.

[Rhea et al., 2005] Rhea, S., Godfrey, B., Karp, B., Kubiatowicz, J., Ratnasamy, S., Shenker, S., Stoica, I., and Yu, H. (2005). Opendht: a public dht service and its uses. In *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 73–84, New York, NY, USA. ACM.

[Röhm et al., 2000] Röhm, U., Böhm, K., and Schek, H.-J. (2000). Olap query routing and physical design in a database cluster. In *EDBT '00: Proceedings of the 7th International Conference on Extending Database Technology*, pages 254–268, London, UK. Springer-Verlag.

[Röhm et al., 2001] Röhm, U., Böhm, K., and Schek, H.-J. (2001). Cache-aware query routing in a cluster of databases. In *Proceedings of the 17th International Conference on Data Engineering*, pages 641–650, Washington, DC, USA. IEEE Computer Society.

[Röhm et al., 2002] Röhm, U., Böhm, K., Schek, H.-J., and Schuldt, H. (2002). Fas: a freshness-sensitive coordination middleware for a cluster of olap components. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 754–765. VLDB Endowment.

[Ronström and Thalmann, 2004] Ronström, M. and Thalmann, L. (2004). Mysql cluster architecture overview - high availability features of mysql cluster. Technical report, MySQL Technical White Paper.

[Rowstron and Druschel, 2001] Rowstron, A. I. T. and Druschel, P. (2001). Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In Guerraoui, R., editor, *Middleware*, volume 2218 of *Lecture Notes in Computer Science*, pages 329–350. Springer.

[Sahin et al., 2004] Sahin, O. D., Gupta, A., Agrawal, D., and Abbadi, E. A. (2004). A peer-to-peer framework for caching range queries. In *ICDE '04: Proceedings of the 20th International Conference on Data Engineering*, pages 165+, Washington, DC, USA. IEEE Computer Society.

[Schütt et al., 2006] Schütt, T., Schintke, F., and Reinefeld, A. (2006). Structured overlay without consistent hashing: Empirical results. In *CCGRID*, page 8. IEEE Computer Society.

[Schütt et al., 2007] Schütt, T., Schintke, F., and Reinefeld, A. (2007). A structured overlay for multi-dimensional range queries. In Kermarrec, A.-M., Bougé, L., and Priol, T., editors, *Euro-Par*, volume 4641 of *Lecture Notes in Computer Science*, pages 503–513. Springer.

[Schütt et al., 2008] Schütt, T., Schintke, F., and Reinefeld, A. (2008). Scalaris: Reliable transactional p2p key/value store. In *ERLANG '08: Proceedings of the 7th ACM SIGPLAN workshop on ERLANG*, pages 41–48, New York, NY, USA. ACM.

[Sit et al., 2004] Sit, E., Dabek, F., and Robertson, J. (2004). UsenetDHT: A low overhead Usenet server. In *Proc. of the 3rd International Workshop on Peer-to-Peer Systems*.

[Stoica et al., 2001] Stoica, I., Morris, R., Karger, D., Kaashoek, F. M., and Balakrishnan, H. (2001). Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, volume 31, pages 149–160, New York, NY, USA. ACM.

[W. W. Stead and Straube, 1983] W. W. Stead, W. E. H. and Straube, M. J. (1983). A chartless record - is it adequate? *Journal of Medical Systems*, 7(2):103–109.

[Walfish et al., 2004a] Walfish, M., Balakrishnan, H., and Shenker, S. (2004a). Untangling the Web from DNS. In *1st Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA.

[Walfish et al., 2004b] Walfish, M., Stribling, J., Krohn, M., Balakrishnan, H., Morris, R., and Shenker, S. (2004b). Middleboxes No Longer Considered Harmful. In *6th Usenix OSDI*, San Francisco, CA.

[Zhao et al., 2001] Zhao, B. Y., Kubiatowicz, J. D., and Joseph, A. D. (2001). Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, EECS Department, University of California, Berkeley.

[Zheng et al., 2006] Zheng, C., Shen, G., Li, S., and Shenker, S. (2006). Distributed segment tree: Support of range query and cover query over dht. In *In Electronic publications of the 5th International Workshop on Peer-to-Peer Systems (IPTPS-06)*.